Incremental Specification and Analysis in the Context of Coloured Petri Nets

by Glenn Anthony Lewis

Computing

Submitted in fulfilment of the requirements
for the Degree of Doctor of Philosophy
University of Tasmania, January 11, 2002

I hereby declare that this thesis contains no material which has been accepted for a degree or diploma by the University of Tasmania or any other institution, except by way of background information duly acknowledged in the thesis, and to the best of my knowledge and belief no material has been previously published or written by another person except where due acknowledgement is made in the text of the thesis.

Glenn Lewis, January 11, 2002

Glenn Lewis, January 11, 2002

The great thing is to last and get your work done and see and hear and learn and understand; and write when there is something that you know ...

ERNEST HEMINGWAY — DEATH IN THE AFTERNOON

Acknowledgements

In an interview for a Ph.D. scholarship I was asked what I thought I would need to successfully complete a Ph.D. My answer focused on the financial and hardware assistance I would need. Indeed such assistance is essential, but as my interviewers pointed out, even more important than financial assistance is support from mentors, family and friends. This thesis would not have been completed without this support.

First and foremost I thank my supervisor Dr Charles Lakos. Charles has guided me throughout my candidature and assisted me greatly in all aspects of my Ph.D. The work presented in this thesis extends earlier work by Charles on Coloured Petri Net refinements and many of the ideas presented in this thesis have resulted from discussion with Charles. The fact that there has been a distance of at least 1000km between our homes for all but the first six months of my candidature has meant that much of this discussion has been by email. That this form of communication has been effective is a testament to Charles' commitment and effort.

When Charles decided to take up a senior lecturing position at the University of Adelaide, Dr Vishv Malhotra agreed to act as my official supervisor at the University of Tasmania. I thank Vishv for this, and for his comments and advice throughout the course of my research.

The algorithms presented in Chapter 7 were implemented in an existing reachability tool. I thank Thomas Mailund and Søren Christensen for their comments on the suitability of the Design/CPN tool, Peter Starke for his comments on the suitability of the INA tool, and Marko Mäkelä for his comments on the suitability of the Maria tool. The Maria tool of the Theoretical Computer Science Laboratory, University of Helsinki, was the tool chosen, and much of the implementation of the algorithms was achieved during an eight month visit to the Laboratory. I thank Nisse Husberg and the other members of the Laboratory for their assistance during this time.

The algorithms were tested on some case studies, including the Missile Simulator study of Steven Gordon and Jonathan Billington. I thank Steven Gordon for supplying a copy of his implementation of the Missile Simulator for comparison with my own implementation.

I am grateful to Nicole Clark and Scott Rayner for proofreading, and Julian Dermoudy for surrendering his computer to me while he was on leave. I also thank the anonymous examiners for their comments helping to improve the quality of the thesis. Finally, I thank my parents for their support and encouragement throughout my entire education, and I thank my friends — Fiona, Bob, Scotty, Disco, Mat, Munna, Stu-D, and everyone from Tae Kwon Do — for their support and for the distractions.

This work has been funded by an Australian Postgraduate Award Scholarship, ARC Large Grant A49800926, and University of Tasmania Department of Computing Top-up Scholarship.

Glenn Lewis,
Hobart,
January 11, 2002

Abstract

Incremental development involves creating a new specification or implementation by modifying an existing one. This is a commonly used technique for handling complex systems in hardware and software engineering. In fact, incremental development is fundamental to *object-orientation*, the widely adopted approach to software engineering which uses the mechanism of *inheritance*.

Incremental development and object-orientation have been adopted for all phases of software engineering, from analysis to design and implementation. In the domain of concurrent systems, some researchers constrain incremental development by proposing requirements that must hold between the original and incrementally modified components. Such proposals are commonly based on a process algebra correctness relation, or require that a bisimulation relation hold between the original and modified components.

In Part I of this thesis we provide background on constraining incremental change and survey several existing proposals. We identify a number of problems typical of these proposals which commonly limit their practical use. We then present *Incremental Coloured Petri Net Modelling* which is aimed at addressing these problems. The main contribution of this part of the thesis is the identification of these problems and the assessment of the practical applicability of Incremental Coloured Petri Net Modelling. This assessment is made by examining several case studies published in the literature.

One of the primary benefits of using a formal method such as Coloured Petri Nets (CPNs) is its support for formal reasoning. State space analysis is a popular formal reasoning technique, but it is subject to state space explosion, where its application to real world models leads to unmanageably large state spaces.

In Part II of this thesis we first review existing approaches for alleviating the state space explosion problem. The main contribution of Part II is a new approach, which we call *Incremental Analysis*. Incremental Analysis involves algorithms which take advantage of Incremental CPN Modelling in attempting to alleviate the state space explosion problem. The thesis considers the implementation issues for these algorithms, identifies the situations under which they can be expected to lead to performance improvement, and presents case studies which demonstrate the value of the technique.

Contents

List of Figures

List of Tables

List of Acronyms

Following is a list of the acronyms that are used in this thesis.

ALU = Arithmetic (and) Logic Unit

ATM = Automatic Teller Machine

BDD = Binary Decision Diagram

CCA = Call Control Application

CCS = Calculus of Communicating Systems

CLOWN = CLass Orientation With Nets

COOPL = Concurrent Object Oriented Programming Language

CO-OPN = Concurrent Object Oriented Petri Nets

CORBA = Common Object Request Broker Architecture

CPN = Coloured Petri Net

CSP = Communicating Sequential Processes

CTL = Computation Tree Logic

DDD = Data Display Debugger

DLE = Data Link Element

DLL = Data Link Layer

DPE = Distributed Processing Environment

EN-system = Elementary Net System

GNU = GNU's Not Unix

GUI = Graphical User Interface

HCPN = Hierarchical Coloured Petri Net

HEC = Hydro-Electric Corporation (of Tasmania)

HLPN = High Level Petri Net

HTTD = Hierarchical Timed Transition Diagram

ISA = International Society for Measurement and Control

ISDN = Integrated Services Digital Network

LAS = Link Access Scheduler

LEN = Labelled Elementary Net

LOOPN = Language for Object Oriented Petri Nets

LOTOS = Language Of Temporal Ordering Specification

LTL = Linear Temporal Logic

LTS = Labelled Transition System

ODP = Open Distributed Processing

OLST = Observable Local State Transformation

OO = Object-Oriented

OOPL = Object-Oriented Programming Language

OOPN = Object-Oriented Petri Net

OOSE = Object-Oriented Software Engineering

OPN = Object Petri Net

PN = Petri Net

PTN = Place Transition Net

RAM = Random Access Memory

RCC = Receiving Call Control

RG = Reachability Graph

RS = Request Substitutability

RNSS = Refined Node State Space

SCC = Strongly Connected Component

SDL = Specification and Description Language

SRM = Sending Resource Management

SSM = State Space Method

ST = State Transformation

STD = State Transition Diagram

SSM = State Space Method

TINA = Telecommunications Informations Networking Architecture

UML = Unified Modelling Language

Chapter 1

Introduction

1.1 Motivation

Concurrent systems are systems composed of elements that can operate concurrently and communicate with each other [79]. The majority of large systems are concurrent [167], and as Meyer says "Concurrency is quickly becoming a required component of just about every type of application, including some which had traditionally been thought of as fundamentally sequential in nature" [140, p. 951].

Concurrent software systems are inherently more complex than sequential software systems. This complexity arises because concurrent systems:

- introduce non-determinism; that is, given the same input and initial state the system can produce several different output states.
- usually exhibit an extremely large number of different behaviours due to the large number of possible interactions between components of the system.
- have design concerns that are not apparent in sequential systems such as locality and synchronisation.

The complexity of concurrent software systems means they are notably difficult to design [205, 140, 79]. To combat the complexity of a concurrent system it is widely recommended that a *formal model* of the system be developed [139, 18, 34, 87, 92, 144]. A formal model is a mathematically based description of the system that allows for reasoning about system properties.

Many formal modelling techniques have been proposed, including process algebras [141, 142, 91, 90, 24, 98], and Petri Nets [158, 102, 164, 78, 3]. Petri Nets are seen to have many positive attributes including: their graphical representation; their even-handed treatment of state and change of state; their ability to be simulated; their well-defined semantics; their ability to be used to represent systems at different levels of abstraction; and their ability to model true concurrency rather than interleaving semantics [145, 185, 102, 162]

Usually the development of a formal model is an incremental process that begins with an abstract model. The abstract model is then refined, possibly over several iterations, to a more concrete model. In fact incremental development is fundamental to *object-orientation* (see Chapter 2), one of the most widely used software engineering methods. In

object-orientation, incremental development is achieved through *inheritance*, where a new specification or implementation is developed by inheriting and incrementally changing an existing specification or implementation. Several proposals have been made to integrate Petri Nets and object-orientation [113, 22, 28, 71, 177].

The integration of object orientation and formal methods has seen a focus on the relationship between the original and incrementally changed component. This has lead some researchers to propose compatibility rules between the original and incrementally changed component [116, 60, 169, 148, 35, 17, 7, 17].

One of the main benefits of developing a formal model is that the model can be *formally analysed.* Here we use the term *formal analysis* in a broad sense to mean answering formal questions about a system's behaviour. Formal analysis therefore encompasses *verification* (mathematically proving if a formal system has a formally stated property), *error detection* (finding errors in a system), as well as general questions such as "What is the maximum number of messages simultaneously in this queue?".

We note that testing (unless it is exhaustive) cannot be used to verify the correctness system [68]. In order to verify (i.e mathematically prove) that a system conforms to a property, all possible behaviours of the system have to be checked.

State based methods are one of the most successful strategies for the formal analysis of concurrent systems. They commonly involve exploring a global state graph representing all behaviours of the system. A major advantage of state based methods is that they are generally automatic and do not require highly skilled personnel.

Formal modelling and analysis is seen as essential in many domains, including *embedded systems*, *safety and security critical systems*, and *protocol development and analysis.* In the following sections we consider these domains in more detail, particularly their need for formal modelling and analysis.

1.1.1 Embedded Systems

An embedded system is one that is physically embedded within a larger system, the primary purpose of which is to maintain some property or relationship between other components of the system [161]. Embedded systems are everywhere, they appear in just about anything electronic including cars, consumer electronics and computer peripherals. It is expected that the number of embedded systems will continue to grow rapidly due to the continued decrease in size and cost of microprocessors, and an increased desire to network systems that have not traditionally been networked.

Embedded systems are often even more complex than non-embedded systems because they must frequently encompass one or more of the following characteristics:

- real-time (the output of the system is dependent on the input and time).
- reactive (the system is in continual interaction with its environment)
- process control (the system controls physical processes and/or mechanical devices).

This complexity together with the fact that embedded systems are often mass produced and cannot be easily changed means that formal methods and the associated formal analysis is often essential. Moreover, in many cases embedded processors are used in mission-critical applications such as medical instrumentation and security systems. A failure here could yield catastrophic results.

1.1.2 Critical Systems

By their very definition, *safety-critical* systems such as aircraft controllers (e.g. Rushby [172]), medical systems (e.g. Ladeau [111]), and nuclear power plant controllers (e.g. Archinoff [14]) must satisfy certain requirements. Formal analysis provides the opportunity to significantly increase confidence that these requirements are satisfied in the final system. For this reason, the use of formal methods has been strongly advocated for such systems [34, 85, 18].

With the explosive growth of the World Wide Web, there is increasing consumer demand for secure access to electronic commerce systems such as money transfer, stock market and electronic shopping. Errors in such security-critical systems are simply not acceptable. To verify the correctness of security-critical components, we must be able to answer the question: is a particular state reachable from the initial state? [157] Formal analysis is the only practical way to answer such questions.

1.1.3 Protocol Development and Analysis

The development of new technologies such as satellite communication, intelligent manufacturing systems, global education products, transport systems and multimedia applications (voice, data, images, video), all rely on various protocols. The consequences of failure of such protocols cannot be overstated; millions of dollars, and/or loss of life could be expected. As Needham et al [146, p. 999] say, protocols are prone to subtle errors that are " ... unlikely to be detected in normal operations. The need for techniques to verify the correctness of such protocols is great ... ". As an example of this, in a paper published in 1996, Lowe [134] breaks the Needham-Schroeder public-key protocol first proposed in 1978. It is often vitally important that protocols are verified to work correctly under all situations. This is something that can only be done using formal modelling and analysis.

1.2 Problem Statement

This thesis addresses two main problems:

> *Problem I:* In the domain of formal modelling for concurrent systems many of the proposals constraining incremental change are aimed at guaranteeing that the incrementally changed component can be substituted for the original component, without the environment being able to detect the difference. Concerns have been raised that such constraints for substitutability are too strong for use in practice [201]. It is not clear what is required for use in practice, and whether a recent proposal by Lakos [116] is appropriate.
>
> *Problem II:* The main and crippling problem of state based formal analysis methods is the often excessive size of the state space. Unfortunately, even for a relatively small model, the size of the state space is often far too large with respect to resources (time and space) to be fully generated [93]. This problem, referred to as the *state space explosion problem*, is the primary obstacle to practical application of State Space Methods (SSMs) [194].

1.3 Contribution of the Thesis

The first part of this thesis examines existing approaches to constraining incremental development and identifies problems with the applicability of such approaches in practice. The approach developed by Lakos [116] — *Incremental CPN Modelling* — is presented and its practical applicability examined. The main contribution of this part of the thesis is the identification of problems typical of existing proposals and the assessment of the practical applicability of Incremental CPN Modelling.

In the second part of this thesis we present algorithms that help alleviate the state space explosion. These algorithms take advantage of each of the forms of incremental change introduced in the first part of the thesis and are referred to as *incremental algorithms*. We discuss the implementation of the incremental algorithms and examine the conditions under which they can be expected to lead to performance improvements. We also examine the performance of the algorithms for some case studies.

It is also worth noting what this thesis does not address. Although this thesis addresses several notions including Object-Orientation, Petri Nets and Coloured Petri Nets, Process Algebra, and equivalence, some notions such as Process Algebra are only introduced at a superficial level for the review of related work. On the other hand Coloured Petri Nets are formally defined and used throughout the thesis. The thesis does not provide a detailed examination of the case studies used to assess the practical applicability of Incremental CPN Modelling. Instead, it presents a summary of the investigations. The thesis provides a review of other state space reduction techniques, but does not provide a detailed examination of the relationship between the incremental algorithms and other state space reduction methods. Further, it does not consider the combination of the incremental algorithms with other state space reduction methods.

1.4 Overview

Chapter 2 presents background information on both parts of the thesis. This chapter covers object-orientation and formalisms for concurrent systems, including an introduction to Coloured Petri Nets. Chapters 3 – 5 constitute Part I of the thesis. Chapter 3 examines existing proposals for constraining incremental change, and identifies several problems typical of such proposals which limit their applicability in practice. Chapter 4 presents Incremental CPN Modelling. By examining a number of case studies from the literature, Chapter 5 assesses the practical applicability of Incremental CPN Modelling.

Chapters 6 – 9 consider the state space explosion problem. Chapter 6 discusses existing approaches to alleviating state space explosion. Chapter 7 presents incremental algorithms which take advantage of the various forms of incremental change to help alleviate state space explosion. Chapter 8 discusses the implementation of the incremental algorithms. In Chapter 9 the performance of the incremental algorithms are compared to that of the standard algorithm for some case studies, and situations under which performance improvement can be expected are identified. Finally, conclusions are drawn and areas for future work are considered in Chapter 10.

Chapter 2

Background

Incremental development is fundamental to object-orientation, one of the most commonly used software engineering methods [183]. The benefits of appropriate incremental development have been widely recognised [183, 200, 140] and several proposals have been made to ensure appropriate incremental change [60, 169, 148, 35, 17, 7, 17]. These proposals often require behavioural compatibility between the original and changed components.

This chapter presents background information relevant to both parts of the thesis. In Section 2.1 we present background information on object-orientation. Section 2.2 introduces Petri Nets, including formal definitions of CPNs, and Section 2.3 discusses behavioural compatibility for concurrent systems. The two parts of the thesis that follow this chapter each present more specific background information and review related work.

2.1 Object-Orientation

Object-oriented technology is not new. It is generally agreed that its inception occurred in the late 1960s with the simulation programming language called Simula, developed by O.J. Dahl and K. Nygaard [61]. Simula introduced the idea of objects simulating real-world entities. In the early 1970s, another programming language, SmallTalk [81], furthered the idea of using software objects to simulate real-world objects for prototyping and developing applications. The 1980s saw a proliferation of so-called object-oriented programming languages. Around this time, faced with this genre of object-oriented programming languages and increasingly complex applications, methodologists began to develop new approaches to analysis and design. Presently many object-oriented analysis and design methodologies have been developed [32, 170, 99, 138, 176, 56, 100], and object-orientation has been incorporated across the spectrum of programming languages, ranging from formal specification languages such as Object-Z [44] and SDL [2], to non-formal implementation languages, such as C++ [5] and Java [15].

Object-orientation is one of the most widely used software engineering methods. Some of the claimed benefits of object-orientation include: improved reuse, increased quality, faster development, and easier maintenance [89, 140]. Object-orientation has made a significant contribution to the solution of problems faced in modern software development and is now providing a basis for researchers to investigate higher levels of abstraction such as design patterns [75] and software architecture [175].

The philosophy underlying the object-oriented paradigm is to consider a system as a collection of active *objects* that collaborate with each other. That is, an object-oriented system is a collection of autonomous concurrent entities, each possessing its own identity, state, and behaviour [32]. The behaviour of each object is implemented by a set of *methods*. Other objects can request that an object — the receiver — behaves in a certain way by requesting that a particular method be performed. Usually only a subset of the receiver's methods can be requested. The names and properties, but not implementations, of this subset of methods can be viewed as the specification of the object, known as the *externally observable behaviour*. The implementation of the methods and any internal data of the object compose its *internal structure*.

A request for an object to perform a particular method is known as a message. Indeed, the only way an object can interact with another object is by sending it a message. Consequently there is a clear separation between the externally observable behaviour of an object and its internal structure. This separation, known as *encapsulation*, means that the internals of the object are "protected" from inadvertent access. Encapsulation is a key feature of an object-oriented system.

An object-oriented system has the potential to evolve by the dynamic creation of objects. To facilitate this, it is common to group objects into *classes*. All the objects of one class have the same internal and external structure and behaviour. A class can be viewed as a template for object creation, or as a set of objects that share a common structure and common behaviour. Objects of a given class are known as *instances* of that class. An instantiation mechanism is provided for the dynamic creation of objects.

A key principle of the object-oriented paradigm that follows from the class concept is the notion of *inheritance*. As Taivalsaari [183, p. 438] says: "Inheritance is often regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, and many of the alleged benefits of object-oriented programming, such as improved conceptual modelling and reusability, are largely accredited to it.".

Inheritance is a technique for using existing class definitions as the basis for new definitions. In the object-oriented paradigm the original class is known as the *parent* or *superclass* while the derived class is called the *child* or *subclass*. Some languages allow a class to be derived from more than one parent class. This is known as *multiple inheritance*.

We adopt Wegner's classification of languages [199], shown in Figure 2.1. Wegner considers a language to be *object-based* if it supports encapsulation. The object-based paradigm requires two enhancements for it to be considered *object-oriented*. The first enhancement introduces the notion of class to the object-based paradigm. This sub-paradigm is therefore known as *class-based*. The second refinement introduces the concept of inheritance, resulting in the *object-oriented* paradigm.

It is common for object-oriented languages to include a *type* system that allows the typing of objects. In general, types impose constraints that help to enforce correctness. In an object-oriented system, types impose constraints on object interaction that prevent objects from inconsistent interaction with other objects [43]. An object-oriented type represents a collection of objects upon which certain predicates hold [201]. A *subtype* of a given type is a new type defined by predicate modification of a given type [201]. We examine the relationship between inheritance and subtyping in Chapter 3.

Another fundamental notion of object-orientation is *polymorphism*. Polymorphism can appear in various situations. Cardelli and Wegner [43] refine the Strachey [182] classification and propose the hierarchy given in Figure 2.2. *Ad-hoc polymorphism* is obtained

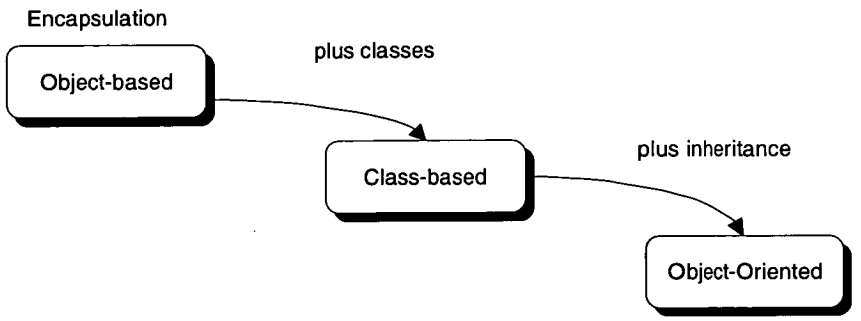


Figure 2.1: The object-based paradigm refined to the object-oriented paradigm (modified from [199, p. 169])

when a function works on several different types (which may not exhibit a common structure) and can behave in an unrelated way for each type. This kind of polymorphism is contrasted to *Universal polymorphism*, in which a function works normally on many types, each of which has a common structure.

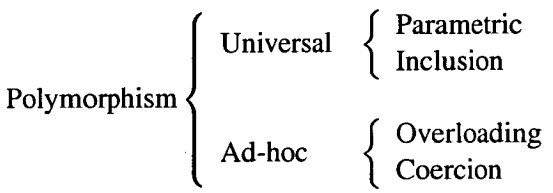


Figure 2.2: Varieties of Polymorphism [43, p. 476]

There are two major kinds of ad-hoc polymorphism. *Overloading* means that the same function name can be used to denote different functions, and then the context determines which function is used. Overloading is merely a syntactic convenience. On the other hand, *Coercion* is a semantic operation that is needed to convert an argument into the type expected by a function in a situation that would otherwise result in a type error.

Universal polymorphism is divided into *Parametric polymorphism* and *Inclusion polymorphism*. In parametric polymorphism a function works uniformly on a range of types that exhibit a common structure. A typical example is the `length` function defined on lists of elements of any type. *Inclusion polymorphism* models subtypes and inheritance. Here an object can be viewed as belonging to many different classes that need not be disjoint; that is, there may be inclusion of classes. *Subtyping* and *subclassing* are examples of inclusion polymorphism.

Soon after the appearance of object-oriented programming languages, methodologists began to develop object-oriented analysis and design methods. By the mid 1990s, a few methods had become clearly prominent, notably, Booch [32], Rumbaugh's OMT [170], Shlaer-Mellor [176], and Jacobson's OOSE [99]. Around this time the primary authors of the Booch, OOSE, and OMT methods joined to develop the Unified Modelling Language (UML) [6], which has rapidly become the de facto industry standard. Booch, Rumbaugh, and Jacobson also provide a development process for use with UML, referred to as the Unified Software Development Process [100]. Another prominent development process, that can be used with UML, is OPEN [84].

UML is a language for visualising, specifying, constructing and documenting the artifacts of a software-intensive system [33]. UML defines nine different types of diagrams for describing different aspects of a system, including a *class diagram* showing classes and their relationships, and *statechart diagrams* showing dynamic behaviour (typically of a class). These diagrams are explained further in Appendix A.

2.2 Formalisms for Concurrent Systems

The design, implementation, and analysis of concurrent systems can be significantly more complex than for sequential systems. The interaction of different parts of the system in different ways means that a concurrent system often has many more possible states than a sequential system. In addition to this, a concurrent system can introduce new kinds of conditions that do not occur in sequential programs, such as deadlock. Formal methods are a way to combat the complexity of a system and reason about the properties of the system. Due to the extra complexity often found in concurrent systems, formal methods for concurrent systems are often essential.

Formal models for concurrent systems generally fall into one of two broad categories: process algebraic models, and state-based models. Process algebra is a widely used framework for describing and reasoning about concurrent systems. Process algebraic models are based on synchronising or communicating processes. That is, the models are a set of processes that execute in parallel, pausing occasionally to synchronise and/or communicate with each other. A process algebraic method provides an algebraic language for the specification of processes and formulation of statements about them, together with calculi for the verification of these statements. Among the best-known process algebras are Milner's CCS [141, 142], Hoare's CSP [91, 90], and Bergstra and Klop's ACP [24]. The International Organisation for Standardization has developed a language known as LOTOS [98] that combines algebraic specification (for data typing) with a process algebra derived from CCS and CSP (for dynamic behaviour).

Bolognesi and Brinksma [31] give a detailed introduction to LOTOS. Here we simply introduce some concepts that are used in a review of related work given in Chapter 3. The process algebra component of LOTOS deals with the description of processes, their behaviour, and how they interact with other processes. It is built on the concepts of observable and hidden behaviour together with the notion of synchronisation. A system is seen as a process, possibly consisting of several sub-processes. An essential component of a process definition is its behaviour expression, which is built by applying an operator to other behaviour expressions. For example, the behaviour expression *a;**stop*** can be built by using the action prefixing operator *;* to prefix the action *a* to the completely inactive process ***stop***. An unobservable action is denoted by ***i***. (Since LOTOS process descriptions are machine readable then ***i*** is used to denote the unobservable action. Formal definitions tend to use τ.) Another operator is the choice operator *[]*. If B_1 and B_2 are two behaviour expressions, then B_1 *[]* B_2 denotes external choice. That is, the environment determines whether the process behaves like B_1 or like B_2. For example, a behaviour expression using choice is: *a;b;c;**stop** [] b;c;a;**stop***. Concurrent cooperation of processes is defined by parallel composition of their behaviour expressions (see [31]).

Another formal model for concurrent systems are Perti Nets. Petri Nets were first defined by Carl Adam Petri [158] and several variations of this first formalism have been

proposed (e.g. [102, 164, 78]). Petri Nets are seen to have many positive attributes including: their graphical representation; their even-handed treatment of state and change of state; their ability to be simulated; their well-defined semantics; their ability to be used to represent systems at different levels of abstraction; and their ability to model true concurrency rather than interleaving semantics [145, 185, 102, 162]. Such positives have meant that Petri Nets have been used effectively for modelling, simulating, and reasoning about concurrent systems in application areas such as: communication protocols, workflow analysis, VLSI chip design, automated production systems, and distributed and embedded systems [104].

Thiagarajan [185, p. 28] lists the guiding principles of Petri nets as:

1. States and change of states (transitions) are two intertwined but distinct notions that deserve an even-handed treatment.
2. Both states and transitions are distributed entities.
3. The extent of change caused by a transition is fixed; it does not depend upon the state at which it occurs.
4. A transition is enabled to occur at a state if and only if the fixed extent of change associated with the transition is possible at that state.

In the literature (e.g. see [165]) it is common to distinguish between a *net* and a *net system*. A net simply defines the static structure of the model, while a net system includes dynamic behaviour. In the following sections we consider a number of types of net systems. The first category of net systems are known as *Low-Level Petri Net Systems* (Section 2.2.1). These include *Elementary Net Systems* [168] and *Place-Transition Net Systems* (PTN systems) [66]. The most significant complaint against low-level net systems is that they are inappropriate for describing complex systems. This has resulted in the development of *High-Level Petri Nets*[1] (HLPNs) such as *Coloured Petri Nets* (CPNs) [102], *Predicate/Transition Nets* (Pr/T Nets) [78], *Algebraic Petri Nets* [3, 30] *Modular CPNs* [25, 102, 72], and more recently *Object-Oriented Petri Nets* (OOPNs) [113, 22, 28, 71, 177]. HLPNs are described in Section 2.2.2.

2.2.1 Low-Level Net Systems

The most basic type of net can be defined as a *bipartite digraph* [189]. Nodes of the graph are either *places* (drawn as ovals) or *transitions* (drawn as rectangles). The directed edges of the graph are the *arcs*, and they connect places to transitions or vice versa.

One of the simplest Petri Net formalisms is *Elementary Net Systems* (EN-systems) [168]. In EN-systems dynamic behaviour is introduced to nets by allowing each place to be marked with a token (drawn as a bold dot). The state of the net system, often referred to as a *marking*, is an indication of whether or not a token is contained in a place at a given point in time. Figure 2.3 shows an EN-system with a token in the place p_1.

Transitions change the distribution of tokens and hence alter the state of the net. Arcs indicate how the transitions affect neighbouring places. The *input places* of a transition

[1] As is common in the literature we refer to HLPNs rather than HLPN Systems. Thus a HLPN includes both static structure and dynamic behaviour.

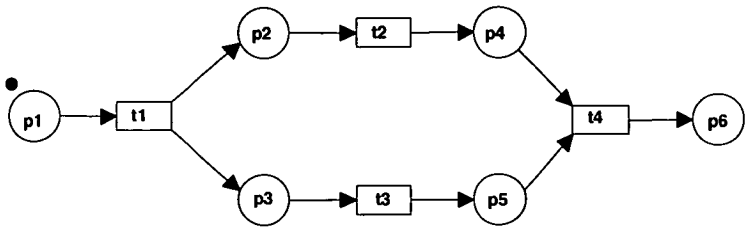


Figure 2.3: A net with one token in place p_1

are those places that have an arc directed from the place to the transition. Similarly the *output places* of a transition are those places that have an arc directed from the transition to the place. A transition is *enabled* if there is a token present in each of its input places. In the net of Figure 2.3 the transition t_1 is *enabled.* An enabled transition can *fire*, which will cause the token to be removed from each of the input places of the transition (in this case the place p_1) and a token to be added to each of the output places of the transition (in this case the places p_2 and p_3), as shown in Figure 2.4. Transitions t_2 and t_3 are then enabled and can fire simultaneously or at separate instants. Transition t_4 will not be enabled until there is a token in both places p_4 and p_5.

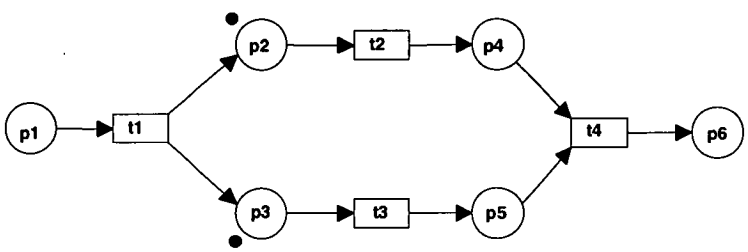


Figure 2.4: The net of Figure 2.3 after transition t_1 fires

The net of Figure 2.5 is an example of an EN-System. It models the dining philosophers problem, which was first proposed by Dijsktra [68]. In the dining philosophers problem five philosophers are seated at a round table with one chopstick between each pair of philosophers and one bowl of spaghetti in the centre of the table. Initially the philosophers are all thinking. At random intervals, each philosopher becomes hungry and decides to eat. This will be possible if the philosopher can get hold of the chopstick on each side; it will not be possible if an adjacent philosopher is eating. We note that in this version of the dining philosophers problem each philosopher picks up both chopsticks simultaneously, thus preventing the situation where some philosophers may only have one chopstick but are not able to pick up the second chopstick as their neighbour has already done so (i.e. this version of the dining philosophers problem does not deadlock).

Place/transition Net Systems (PTN-systems) are the most prominent and best studied class of Petri Nets [66]. PTN-systems are similar to EN-systems except that places can store more than one token. Formal definitions of basic PTN-systems and common extensions can be found in [66]. One such common extension to basic PTN-systems are *arc weights.* A weighted arc specifies that more than one token is removed from a place, or added to a place by a single occurrence of a transition.

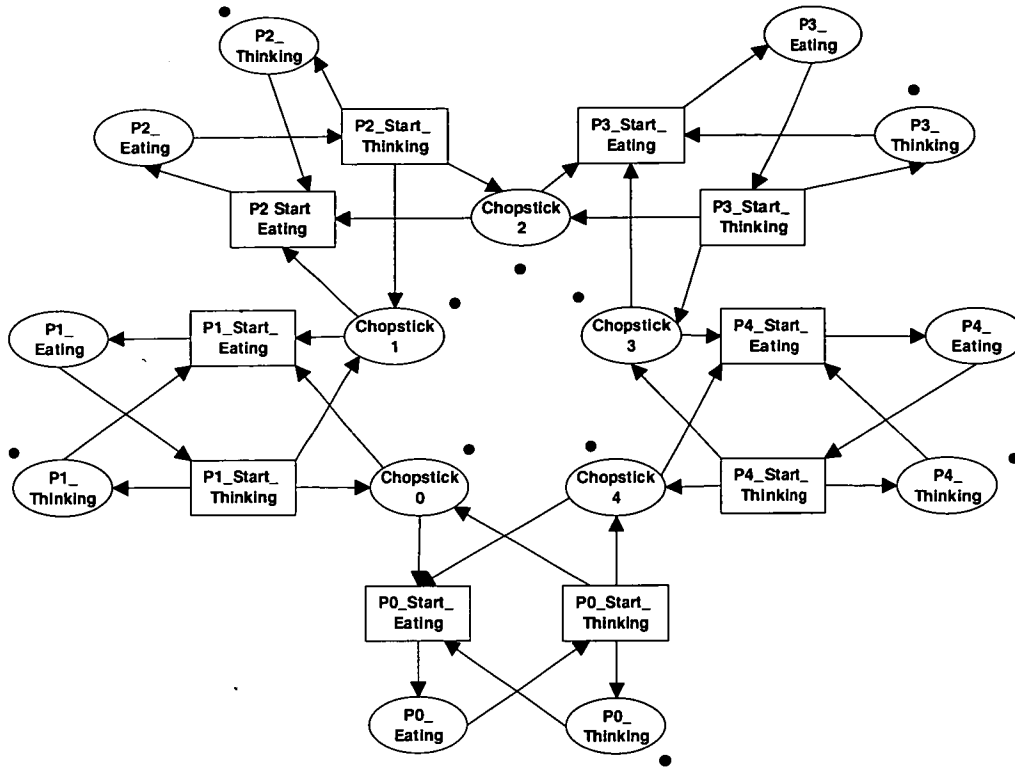


Figure 2.5: Dining Philosophers Elementary Net System

2.2.2 High-Level Petri Nets

High-Level Petri Nets (HLPNs) are an extension of PTN Systems. Most practical applications use one of the different kinds of HLPNs. Basic kinds of HLPNs include *Algebraic Petri Nets* [3, 30], *Predicate/Transition Nets* (Pr/T Nets) [78] and *Coloured Petri Nets* (CPNs) [102]. Typically HLPNs do not have any greater modelling power than PTN systems — every Pr/T net or CPN has an equivalent PTN. However, HLPNs provide a greater descriptive power by avoiding much of the duplication that is necessary in PTNs.

In our formulation of CPNs (see below) each place and transition is typed by a colour set. A token is a value taken from the type of its associated place. An arc from a place to a transition or vice versa can be annotated with a function over the colour of the transition. Each arc function maps a colour (or *firing mode*) of a transition to a multiset over the colour of the place.

A CPN model of the dining philosophers (see Section 2.2.1), is given in Figure 2.6. In this net addition is assumed to be modulo 5. With n philosophers the CPN only needs 3 places and 2 transitions, whereas the behaviourally equivalent PTN would require $3n$ places and $2n$ transitions [190]. The Design/CPN tool [105] is one of the most widely used tools supporting the use of CPNs. The notation adopted for CPN figures in this thesis is similar to the Design/CPN notation. Thus the colour of each place is written in italics next to the place. In the dining philosophers CPN, the colour of each place is *Int5*, where *Int5* is declared to be the set $\{0,1,2,3,4\}$. A common notation [102] for multisets is adopted. This involves representing the multiset as a sum, with the number of appearances of each element of the set indicated before the element. For example, the multiset $1`a + 2`b$ is

the multiset with one a element and two b elements. In the CPN figures presented in this thesis, the initial marking of each place that is not empty is the multiset sum written next to the place (note that opposed to the Design/CPN notation, we do not underline initial markings). In the dining philosophers CPN, the initial marking of the place *Chopsticks* is $1`0 + 1`1 + 1`2 + 1`3 + 1`4$. The firing modes (colours) of each transition are not explicitly given in the net description. Instead, each arc is annotated with an expression involving one or two variables. The set of firing modes (or colour) of a given transition is assumed to be the cross product of the colour sets of the places input to the transition. The set of firing modes for a transition can be restricted by an expression that is given in square brackets next to the transition. This condition is referred to as the *guard* of the transition. The guard of the *eat* transition of the dining philosophers CPN is $[j = i + 1]$. (Recall that addition is assumed to be modulo 5.) Note that this guard is artificial and would not normally be used in this context, as is the case in the *think* transition, but is included to illustrate a guard.

As with a PTN system, the marking of a CPN consists of the marking of each place of the net and describes the global state of the system. In order for a state to change a step must occur. Each step consists of one or more transitions each with a particular firing mode. For example, given the initial marking of the dining philosophers CPN as shown in Figure 2.6, the transition *eat* can occur with the mode 0 where i will be equal to 0 and j will be equal to 4 (since $i = 4$, then $(i+1) \bmod 5 = 0$). This will result in the multiset sum $1`0 + 1`4$ being removed from the place *Chopsticks*, $1`0$ being removed from the place *Thinking_Philosophers*, and $1`0$ added to the place *Eating_Philosophers*.

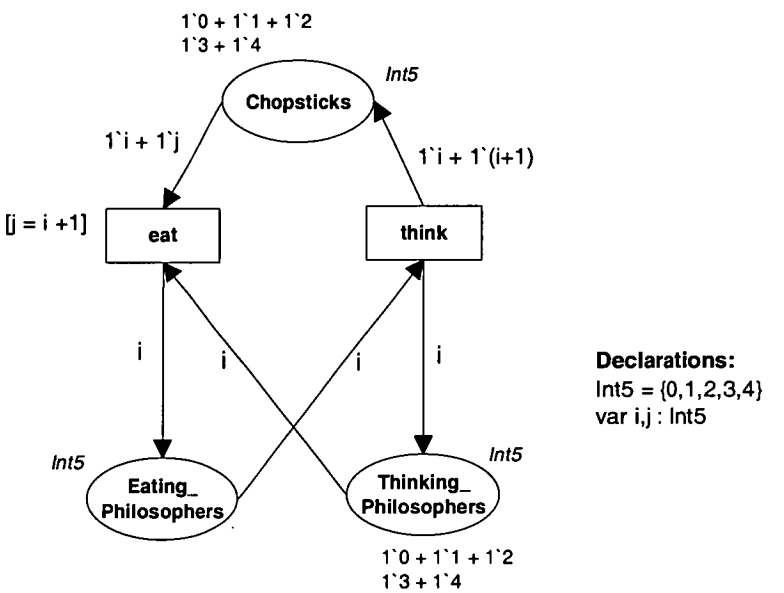


Figure 2.6: Dining Philosophers Coloured Petri Net

We now formally define CPNs. The following definitions and explanatory notes are taken from Lakos [116]. This formulation of CPNs is slightly different from the common formulation [102], but they are to all intents and purposes equivalent. It is convenient to use this formulation in later chapters.

For completeness the Definitions 2.1 – 2.3 define mathematical preliminaries (functions, binary relations, multisets, and relations and operations on multisets). These preliminaries are standard mathematical definitions. We use $\mathbb{N}$ to denote the natural numbers, that is $\mathbb{N} = \{0,1,2,\ldots\}$. The *power set* of a set A, denoted by 2^A, is the set of all subsets of A.

Definition 2.1. For two nonempty sets A and B, a *function* f mapping A to B, written $f : A \to B$, is a subset of $A \times B$ such that for each $a \in A$, there is one and only one element $b \in B$ such that $(a,b) \in f$. For a function $f : A \to B$ the domain of f is $domain(f) = A$, and the range of f is $range(f) = \left\{ y \;\middle|\; y \in B \;\wedge\; y = f(x) \text{ for some } x \in A \right\}$. A function $f : A \to B$ is:

injective if $f(a_1) \neq f(a_2)$ whenever $a_1 \neq a_2$,

surjective if for every $b \in B$ there exists $a \in A$ such that $f(a) = b$,

An injective and surjective function is called a *bijection.* For every set A the identity function $Id : A \to A$ is given by $Id(x) = x$ for all $x \in A$. Clearly Id is a bijection. A bijection from a set A to itself is called a *permutation* on A. For any nonempty set A, we adopt the notation $\pi(A)$ for the set of all permutations on A.

Definition 2.2. A *relation* (or a *binary relation*) on a nonempty set A is a nonempty set R of ordered pairs (a,b) of elements $a,b \in A$. A relation $R \subseteq A \times A$ is:

reflexive if $\forall a \in A \;\; (a,a) \in R$.

transitive if $(a,b) \in R \;\wedge\; (b,c) \in R \Rightarrow (a,c) \in R$

symmetric if $(a,b) \in R \Rightarrow (b,a) \in R$

A reflexive and transitive relation is a *preorder*, while a reflexive, transitive, and symmetric relation is an *equivalence*.

Definition 2.3. A *multi-set*, β, (also known as a *bag*) over a non-empty *basis* set A is a function:

$$\beta : A \to \mathbb{N}$$

We usually represent the multiset β by a formal sum:

$$\sum_{a \in A} \beta(a) \grave{} a$$

We define the *multisets* over A as $\mu A = \{\beta \mid \beta : A \to \mathbb{N}\}$. Let A be a non-empty set, $\beta_1, \beta_2 \in \mu A$, and $n \in \mathbb{N}$. As in [3, 137] we define the following relations and operations:

a. $a \in \beta \Leftrightarrow \beta(a) > 0$ (membership)

b. $\beta_1 = \beta_2 \Leftrightarrow \forall a \in A : \beta_1(a) = \beta_2(a)$ (equality)

c. $\beta = \emptyset \Leftrightarrow \forall a \in A : \beta(a) = 0$ (empty multiset)

d. $\beta_1 \leq \beta_2 \Leftrightarrow \forall a \in A : \beta_1(a) \leq \beta_2(a)$ (containment)

e. $\beta = \beta_1 + \beta_2 \Leftrightarrow \forall a \in A : \beta(a) = \beta_1(a) + \beta_2(a)$ (addition)

f. $\beta = \beta_1 - \beta_2 \Leftrightarrow \forall a \in A : (\beta_1(a) \geq \beta_2(a)) \wedge \beta(a) = \beta_1(a) - \beta_2(a)$ (subtraction)

g. $\beta = n\beta_1 \Leftrightarrow \forall a \in A : \beta(a) = n \times \beta_1(a)$ (scalar multiplication)

h. $|\beta| = \sum_{a \in A} \beta(a)$ (cardinality)

Coloured Petri Nets

We define Coloured Petri Nets in the context of a *universe of non-empty colour sets* Σ with an associated *partial order* $<: \subseteq \Sigma \times \Sigma$ that is derived from type compatibility in the theory of object-oriented languages [42]. $X <: Y$ means that the values of X can be used in contexts expecting values of Y, and typically that X has extra data components over Y. In this case we assume the existence of a (polymorphic) projection function Π_Y from the values of X to those of Y (that do not appear in any proper subtype). For our purposes, we only use the fact that values of X are also values of Y.

Definition 2.4. As in [116, Section 3.1], given a universe of colour sets Σ we define:

a. the *functions* over Σ as $\Phi\Sigma = \{X \to Y \mid X, Y \in \Sigma\}$,

b. the *sequences* over a non empty set X as $\sigma X = \{x_1 x_2 \ldots x_n \mid (n \geq 0) \wedge (x_i \in X)\}$ together with the empty sequence. For $S^* = s_1 s_2 \ldots s_n \in \sigma X$ we use $\#(S^*)$ to denote the number of elements in the sequence and $s_1 \in S^*$ to indicate that s_1 is a member of the sequence S^*. Further, the function $head(S^*)$ returns the head of S^*, that is it returns s_1, and the function $tail(S^*)$ returns the tail of S^*, that is it returns $s_2 \ldots s_n$. We usually name sequences with an asterix superscript.

We now define CPNs. Here we provide a definition of the underlying semantics rather than of the graphical form that was introduced in Section 2.2.2. In order to clarify this distinction see [3].

Definition 2.5 (Coloured Petri Net). [116, def. 3.1]
A *Coloured Petri Net* N is a tuple $N = (P, T, A, C, E, \mathbb{M}, \mathbb{Y}, M_0)$ where:

a. P is a set of places

b. T is a set of transitions, s.t. $P \cap T = \emptyset$

c. A is a set of arcs, s.t. $A \subseteq (P \times T) \cup (T \times P)$

d. $C : P \cup T \rightarrow \Sigma$ determines the colours of places and (modes) of transitions

e. $E : A \rightarrow \Phi\Sigma$ gives the arc inscriptions, s.t. $E(a) : C(t) \rightarrow \mu C(p)$

f. $\mathbb{M} = \mu\{(p,c) \mid p \in P,\ c \in C(p)\}$ is the set of markings

g. $\mathbb{Y} = \mu\{(t,c) \mid t \in T,\ c \in C(t)\}$ is the set of steps

h. M_0 is the initial marking, $M_0 \in \mathbb{M}$

Note that there is at most one arc in each direction for any (place,transition) pair and that the effect of an arc is given by the arc inscription in conjunction with a particular transition firing mode. We refer to a (place,colour) pair as a *token element*, and a (transition,colour) pair as a *firing element*. The set of all token elements of a CPN N is denoted by TE:

$$TE = \left\{ (p,c) \;\middle|\; p \in P,\ c \in C(p) \right\}$$

The set of all firing elements of a CPN N is denoted by FE:

$$FE = \left\{ (t,c) \;\middle|\; t \in T,\ c \in C(t) \right\}$$

We define $FE|_X \subseteq FE$ as the restriction of FE to firing elements involving transitions in $X \subseteq T$. That is $FE|_X = \{(t,c) \in FE \mid t \in X\}$.

Therefore the markings of N are multisets of token elements, and the steps of N are multisets of firing elements. While markings and steps are derivative quantities, they are included in the definition of a CPN so that it is clear that system morphisms $\phi : N \rightarrow N'$ (introduced in Chapter 4) map markings and steps to markings and steps respectively.

As Lakos says [116, p.329], Definition 2.5 is "to all intents and purposes, equivalent to the common definition [102]". It does not include a guard function defined on transitions, but the same effect is achieved by limiting the colour set associated with the transition.

Definition 2.6. For a CPN N, $x \in P \cup T$, we define:

the inputs of x, ${}^\bullet x = \{y \in P \cup T \mid (y,x) \in A\}$

the outputs of x, $x^\bullet = \{y \in P \cup T \mid (x,y) \in A\}$

Having defined the structure of CPNs, we are now ready to define their behaviour.

Definition 2.7. [116, def. 3.3]
The *incremental effects* $E^+, E^- : \mathbb{Y} \to \mathbb{M}$ of the occurrence of a step Y are given by:

$$E^-(Y) = \sum_{(t,m) \in Y} \sum_{(p,t) \in A} \{p\} \times E(p,t)(m)$$

$$E^+(Y) = \sum_{(t,m) \in Y} \sum_{(t,p) \in A} \{p\} \times E(t,p)(m)$$

Definition 2.8. [116, def. 3.4]
For a CPN, N, a step $Y \in \mathbb{Y}$ is *enabled* in marking $M \in \mathbb{M}$, written $M[Y\rangle$, if and only if $M \geq E^-(Y)$.

Definition 2.9. [116, def. 3.4]
If a step $Y \in \mathbb{Y}$ of CPN N is enabled in marking $M_1 \in \mathbb{M}$, it can *fire* leading to marking $M_2 \in \mathbb{M}$, written $M_1[Y\rangle M_2$ with $M_2 = M_1 - E^-(Y) + E^+(Y)$.

Definition 2.10. [116, def. 3.4]
A step sequence $Y^* = Y_1 Y_2 \cdots Y_n \in \sigma\mathbb{Y}$ of a CPN N is enabled in marking $M_1 \in \mathbb{M}$ and can occur, leading to marking $M_2 \in \mathbb{M}$, written $M_1[Y^*\rangle M_2$, if there exists intermediate markings $M_2', M_3', \ldots, M_n' \in \mathbb{M}$ such that:

$M_1[Y_1\rangle M_2'$ and

$M_i'[Y_i\rangle M_{i+1}'$ for $i \in 2, \ldots, n-1$ and

$M_n'[Y_n\rangle M_2$

Definition 2.11. [116, def. 3.4]
We use $[M\rangle$ to denote the set of markings reachable from $M \in \mathbb{M}$. That is:

$$[M\rangle = \{M_1 \in \mathbb{M} \mid \exists\, Y^* \in \sigma\mathbb{Y} : M[Y^*\rangle M_1\}$$

Definition 2.12. [116, def. 3.5]
For CPN N step $Y \in \mathbb{Y}$ is *realisable* by $Y^* \in \sigma\mathbb{Y}$ in marking $M_1 \in \mathbb{M}$ leading to marking $M_2 \in \mathbb{M}$ if $M_1[Y^*\rangle M_2$ and $\sum_{y \in Y^*} y = Y$.

Definition 2.13. [116, def. 3.6]
For a CPN N, the set of *reachable markings* $\mathbb{M}_R \subseteq \mathbb{M}$ is given by:

$$\mathbb{M}_R = \left\{ M \in \mathbb{M} \;\middle|\; \exists\, Y^* \in \sigma\mathbb{Y} : M_0[Y^*\rangle M \right\}$$

Definition 2.14. [116, def. 3.6]
For a CPN N, the set of *enabled step sequences* ${\mathbb{Y}^*}_E \subseteq \sigma\mathbb{Y}$ is given by:

$$\mathbb{Y}_E{}^* = \left\{ Y^* \in \sigma\mathbb{Y} \;\middle|\; \exists M \in \mathbb{M} : M_0[Y^*\rangle M \right\}$$

A range of other high level net formalisms have been proposed, including Algebraic Nets [30, 164, 3] and Predicate Transition Nets (Pr/T Nets) [78]. An Algebraic Net consists of a coloured net where colours and firing rules are represented by interpretations of terms over an algebraic specification. Pr/T Nets were the first class of HLPNs proposed and are very similar to CPNs. (There are some technical differences between Pr/T Nets and CPNs.) CPNs are the most widely used HLPN formalism. Most of the other HLPN formalisms have a similar descriptive power to that of CPNs, and differ only in their underlying formalism.

The absence of compositionality has been one of the major criticisms raised against Petri Net models [102]. This has led to the development of Modular Petri Nets (Section 2.2.3) and Object-Oriented Petri Nets (Section 2.2.4).

2.2.3 Modular Petri Nets

Modular Petri Nets [25, 102, 72] are a class of Petri Nets that allow a net to be described as a set of interacting modules. Typically the modules interact by shared places and/or shared transitions. Sharing is often accomplished by *fusion sets*. A *place fusion set* is a set of places that are considered to be a single conceptual place, while a *transition fusion set* is a set of transitions that are considered to be a single conceptual transition.

Hierarchical Coloured Petri Nets (HCPNs) [102] are CPNs extended to provide hierarchy constructs, whereby a subnet can be isolated as a separate page. HCPNs allow (coloured) nets to be constructed from CPN modules using fusion places. Another construct, *substitution transitions*, is also used to provide hierarchy in HCPNs. Substitution transitions allow an abstract transition to be drawn in a top level diagram of the net and a more concrete net to be drawn for that transition in a lower level diagram. Substitution transitions are essentially macro expansions that do not require behavioural compatibility between the net before substitution of the transition and the net after substitution of the transition.

2.2.4 Object-Oriented Petri Nets

Both Petri Nets and the object-oriented paradigm are seen to have strengths and weaknesses. Petri Nets are a formal model for concurrent systems that possess an easy to understand graphical representation. However, in their basic form Petri Nets are not modular, meaning that it is not possible to produce a model of an industrially-sized system that is easy to understand. On the other hand, object-orientation was introduced pragmatically and without a formal basis. At the core of object-orientation is modularity.

It is not surprising, therefore, that in order to provide powerful structuring and compositional mechanisms for Petri Nets, and/or to provide a formal basis for object-orientation, several proposals have been made to integrate Petri Nets and object-orientation [113, 22, 28, 71, 177]. As Bastide [20] says, there are two main ways in which object-orientation has

been integrated into Petri Nets. The first of these is the *Objects Inside Petri Net* approach. Here one overall net acts as a control structure and the tokens of the net are objects which constitute the data structures of the net. The second approach is called *Petri Nets Inside Objects*. Here the system is divided into several objects each of which uses a Petri Net to describe its internal behaviour. A more general approach to integrating object-orientation and Petri Nets encompasses both of the previous approaches. This approach supports both *Petri Nets inside objects* and *Objects inside Petri Nets*.

2.3 Behavioural Compatibility for Concurrent Systems

As will be considered in more detail in Chapter 3, it has been argued that there should be a behavioural relationship between an original object and an incrementally changed version of that object. In an approach due to Milner [141], the properties of two concurrent systems are related by their observable behaviour. Analogous with observations made by humans when comparing systems, the more power the observers of concurrent systems have to see the details of a system, the more distinctions can be made between the systems. Various equivalence or preorder relations have been proposed for comparing concurrent systems, each of which has different capabilities for making distinctions between systems. If one relation makes fewer distinctions than another it is said to be a *weaker* relation.

There has been significant work on equivalences and preorder relations in the context of concurrent systems. van Glabbeek [196] reviews 155 equivalences based on the observation of actions. Pomello et al [160] survey behavioural compatibility for elementary Petri Nets. They identify a broad group of relations based on observation of actions, many of which have been adapted from other theories of concurrent systems. This group can be further subdivided depending on whether the net semantics is interleaved, step-based, or partial order. The duality between places and transitions characterising Petri Nets also allows Pomello et al to identify a second group of relations — those based on observation of states.

Historically, such equivalence and preorder relations are used as criteria for determining whether implementations (say of a protocol) meet specifications. More recently, however, proposals have required that various equivalence and preorder relations hold for incremental change (particularly in the context of object-oriented languages).

van Glabbeek [195] has proposed the linear/branching time spectrum as a unifying framework for classifying action based observable equivalences. Figure 2.7, taken from [195, p. 280], illustrates the classification as a hierarchy for 11 concurrency semantics which are uniformly definable in terms of action relations. An arrow from a semantics R to a semantics Q means that R is stronger than Q. There are many other forms of equivalence in [195] and elsewhere which have not been included in Figure 2.7.

Typically such relations are defined in the context of *Labelled Transition Systems* [202, 195, 193]. It will be clear that by attaching a label to each transition of an Elementary Petri Net, the same relations can be defined for Elementary Petri Nets. The definition of such relations for higher level nets (such as CPNs) is not as simple (see Section 4.7). We now define Labelled Transition Systems. The following definitions are adapted from those of Wehrheim [202]. They are used in Chapter 4, Section 4.7 when examining the relationship of Incremental CPN modelling [116] with other equivalence relations.

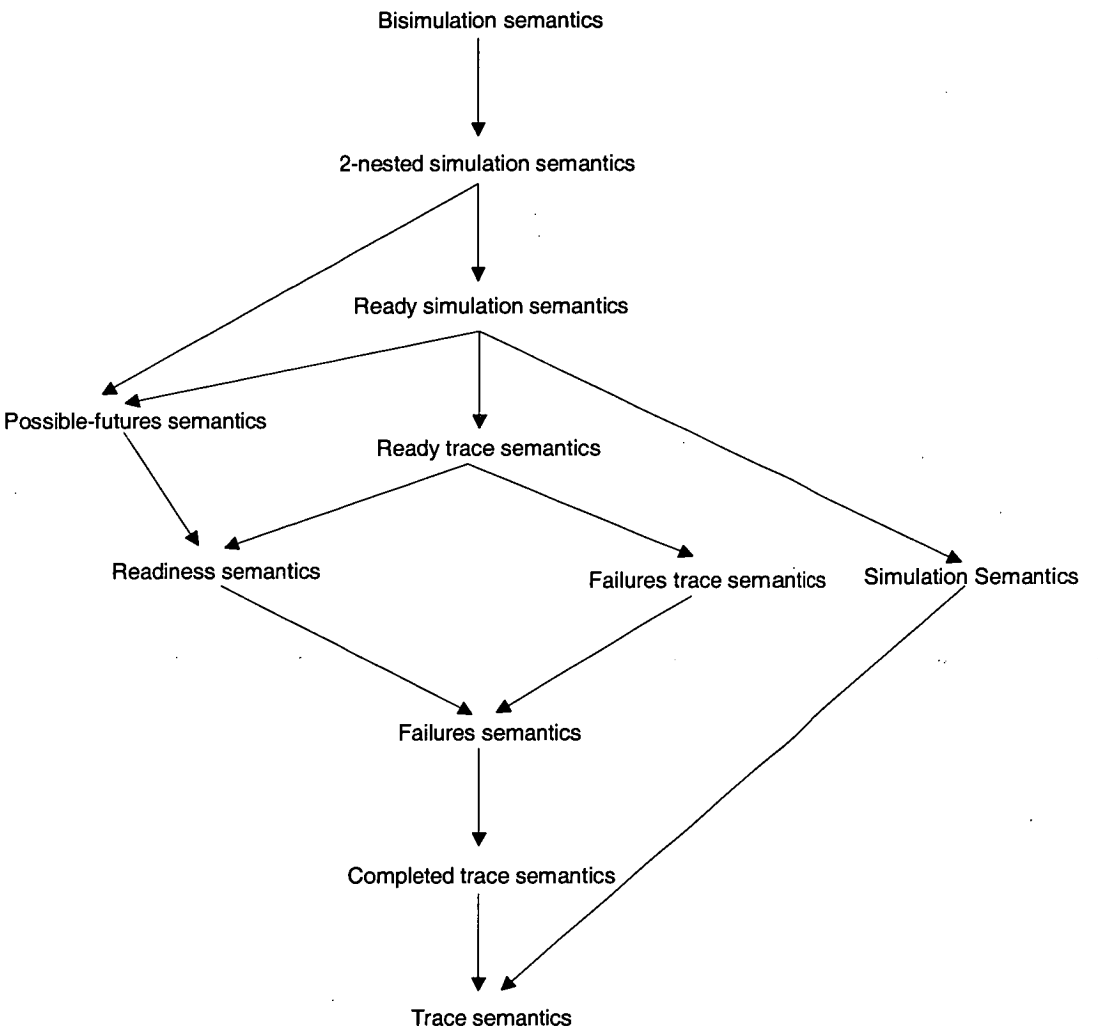


Figure 2.7: The linear-time/branching hierarchy [195, p. 280]

Let Λ be a set of actions, and τ be a special *invisible* action, $\tau \notin \Lambda$, $\Lambda_\tau = \Lambda \cup \{\tau\}$. $\sigma\Lambda$ is the set of all sequences over Λ, and $\sigma\Lambda_\tau$ is the set of all sequences over Λ_τ.

Definition 2.15. A *Labelled Transition System* (LTS) is a tuple $T = (\Lambda_\tau, Q, \rightarrow, q_0)$ such that:

a. Q is a set of states

b. $\longrightarrow \subseteq Q \times \Lambda_\tau \times Q$ the transition relation

c. $q_0 \in Q$ is the initial state

Definition 2.16. Let $\alpha^* \in \sigma\Lambda_\tau$, $A \subseteq \Lambda_\tau$, and $a_i \in \Lambda_\tau$ for $0 < i \leq n$. The *projection* of α^* on A, $\alpha^* \restriction A$, is the trace where all occurrences of actions not in A are deleted. We write:

- $q \xrightarrow{a_1} q'$ if $(q, a_1, q') \in \longrightarrow$.
- $q \xrightarrow{a_1 \ldots a_n} q'$ if there are states $q_0, q_1, \ldots, q_n$ such that $q = q_0$, $q_i \xrightarrow{a_{i+1}} q_{i+1}$ and $q_n = q'$
- $q \stackrel{\alpha^*}{\Longrightarrow} q'$ iff there is a trace $t^* \in \sigma\Lambda_\tau$ such that $q \xrightarrow{t^*} q'$ and $\alpha^* = t^* \restriction \Lambda$.

Definition 2.17. The set of traces of an LTS $T = (\Lambda_\tau, Q, \rightarrow, q_0)$ is:

$$traces(T) = \{\alpha^* \in \sigma\Lambda \mid \exists\, q \in Q : q_0 \stackrel{\alpha^*}{\Longrightarrow} q\}$$

Two systems T_1 and T_2 are trace equivalent iff $traces(T_1) = traces(T_2)$.

In general, trace equivalence does not preserve deadlock situations [160]. The notion of failure equivalence, introduced by Hoare [91] in defining the semantics of CSP, was motivated by the perceived need to find an equivalence relation that preserves information on deadlocks but is otherwise as weak as possible. Two systems are failure equivalent if they have the same failure to observe a specific action after having made the same sequence of observations.

A state is *stable* if no τ actions are possible. The set of *enabled actions* of a stable state $q \in Q$ is: $enabled(q) = \{a \in \Lambda \mid \exists\, q' \in Q : q \xrightarrow{a} q'\}$. The maximal *refusals* of a stable state $q \in Q$ are $refusals(q) = \Lambda \setminus enabled(q)$

Definition 2.18. The set of failures of an LTS $T = (\Lambda_\tau, Q, \longrightarrow, q_0)$ is:
$failures(T) = \{(\alpha^*, X) \in \sigma\Lambda \times 2^\Lambda \mid \exists\, q \in Q : (q_0 \stackrel{\alpha^*}{\Longrightarrow} q) \wedge (q \textit{ is stable}) \wedge (X \subseteq refusals(q))\}$
Two systems T_1 and T_2 are failure equivalent iff $failures(T_1) = failures(T_2)$.

Two systems are *bisimulation equivalent* if after having observed the same sequence of actions on both systems, they are henceforth bisimilar.

Definition 2.19. Given two labelled transition systems $T_1 = (\Lambda_\tau, Q_1, \longrightarrow_1, q_{01})$ and $T_2 = (\Lambda_\tau, Q_2, \longrightarrow_2, q_{02})$ a binary relation $R \subseteq Q_1 \times Q_2$ is a *bisimulation* if for all $(q_1, q_2) \in R$ the following two conditions hold:

a. for each $\alpha^* \in \sigma\Lambda_\tau$ and $q_1' \in Q_1$ s.t. $q_1 \stackrel{\alpha^*}{\Longrightarrow} q_1'$ there is $q_2' \in Q_2$ s.t. $q_2 \stackrel{\alpha^*}{\Longrightarrow} q_2'$ and $(q_1', q_2') \in R$

b. for each $\alpha^* \in \sigma\Lambda_\tau$ and $q_2' \in Q_2$ s.t. $q_2 \stackrel{\alpha^*}{\Longrightarrow} q_2'$ there is $q_1' \in Q_1$ s.t. $q_1 \stackrel{\alpha^*}{\Longrightarrow} q_1'$ and $(q_1', q_2') \in R$

Two LTSs are *bisimilar* if there is a bisimulation relating their initial markings (i.e. $(q_{01}, q_{02}) \in R$).

The above definition is often referred to as *weak bisimulation* since the set of actions includes the invisible action τ. The definition of *strong bisimulation* is the same as weak bisimulation, but the set of actions does not contain the special invisible action and the relation $\stackrel{\alpha^*}{\Longrightarrow}$ is replaced with $\xrightarrow{\alpha^*}$.

Pomello et al [160] also consider equivalence of elementary net systems based on observation of states. Here a number of places are designated as observable and markings of the net that consist solely of observable places are called observable.

2.4 Summary

In this chapter we have supplied the background for the subsequent two parts of the thesis. The first part of the chapter presented objected orientation, including an introduction to inheritance, polymorphism, and object-oriented methods. Section 2.2 presented background information on formalisms for concurrent systems, with particular emphasis on Petri Nets, including a formal definition of Coloured Petri Nets. Finally, Section 2.3 considered behavioural compatibility for concurrent systems. Here Labelled Transition Systems were defined, and various equivalence relations including trace equivalence, failures equivalence, and bisimulation were discussed.

Part I

Incremental Specification

Chapter 3

Existing Approaches to Incremental Change

Of all issues in object technology, none causes as much discussion as the question of when and how to use inheritance; sweeping opinions abound, for example on Internet discussion groups, but the literature is relatively poor in precise and useful advice.
BERTRAND MEYER [140, p. 809]

There have been several proposals [60, 169, 148, 35, 17, 7, 17] for constraining incremental change. In this chapter we give the general background to the various approaches proposed and survey some of these proposals. An abbreviated version of this survey has already been published [124]. We discuss why these proposals are rarely adopted in practice.

3.1 Appropriate Incremental Development

In object-oriented systems, incremental development is achieved using *inheritance* (see Section 2.1), and therefore much of the existing work on appropriate incremental change has been formulated in the context of object-oriented systems. As we noted in Chapter 2, many of the benefits of object-oriented programming are largely credited to inheritance.

One of the main benefits of inheritance is due to its support for abstraction, which is the most common and effective technique for dealing with complexity [68]. In particular inheritance provides support for *conceptual specialisation*, which as Taivalsaari [183] explains, involves the ordering of knowledge into hierarchies of abstractions. Another main benefit of inheritance is that it provides support at the implementation level for the reuse of code. That is, existing classes can be used as the basis for the definition of new classes.

Historically, a common belief was that the use of inheritance for conceptual specialisation should coincide with the use of inheritance for implementation. However towards the mid-to-late 1980s a number of researchers expressed the idea that a clear distinction ought to be made between the two concepts of subclassing and subtyping [178, 9, 57, 126]. *Subclassing* is a low level mechanism by which classes can share behaviour or data. *Subtyping*, on the other hand, was said to express specialisation. Pierre America explains the distinction:

> ... it is useful to distinguish between the notions of class and type ... a class as a collection of objects that have the same internal structure: the same variables, methods and body. Then we can use the term "type" for a collection of objects that have certain common properties with respect to their behaviour. In other words, whereas a class groups together objects that have been built in the same way, a type comprises a collection of objects that can be used in a certain way. [10, p. 395].

LaLonde and Pugh [126] argue that similar to the distinction between inheritance and subtyping, there is a significant difference between subtyping and specialisation. They have proposed the following three definitions:

- *Subclassing* is an implementation mechanism for sharing code and representation.
- *Subtyping* is a substitutability relationship: an instance of a subtype can replace an instance of its supertype.
- *Is-a* is a conceptual specialisation relationship: it describes one kind of object as a special kind of another.

LaLonde and Pugh suggest that subclassing, subtyping, and specialisation are all important for different reasons [126]. Subclassing supports reusability for the class library implementor: new classes can be based on existing classes. Subtyping supports reusability for the class library user: to get maximum reusability we need to know which classes can be substituted by which others. Specialisation relationships are important for understanding the relationships between the concepts. In this sense, specialisation is important for the class library designer.

Figure 3.1 illustrates the differences between subclassing, subtyping, and specialisation. This shows, for example, that a *Set* is not a subtype of a *Bag*, since a *Set* cannot be substituted for a *Bag* (as operations on the *Bag* may assume duplicate elements exist). However, a *Set* is a conceptual specialisation of a *Bag*, since a *Set* eliminates duplicates in a *Bag*.

The above definitions indicate that inheritance is a mechanism that is suited but not necessarily limited to specialisation. Similarly, inheritance is a mechanism that can be used to develop subtypes, but not all classes developed using inheritance are necessarily substitutable. The value of inheritance as a mechanism to facilitate abstraction and conceptual specialisation has been widely recognised by practitioners for all phases of object-oriented system design, from analysis to design and implementation [183, 173, 140]. On the other hand, the use of inheritance merely for implementation purposes alone is likely to cause difficulties and reflects poor understanding of the purpose of inheritance [183].

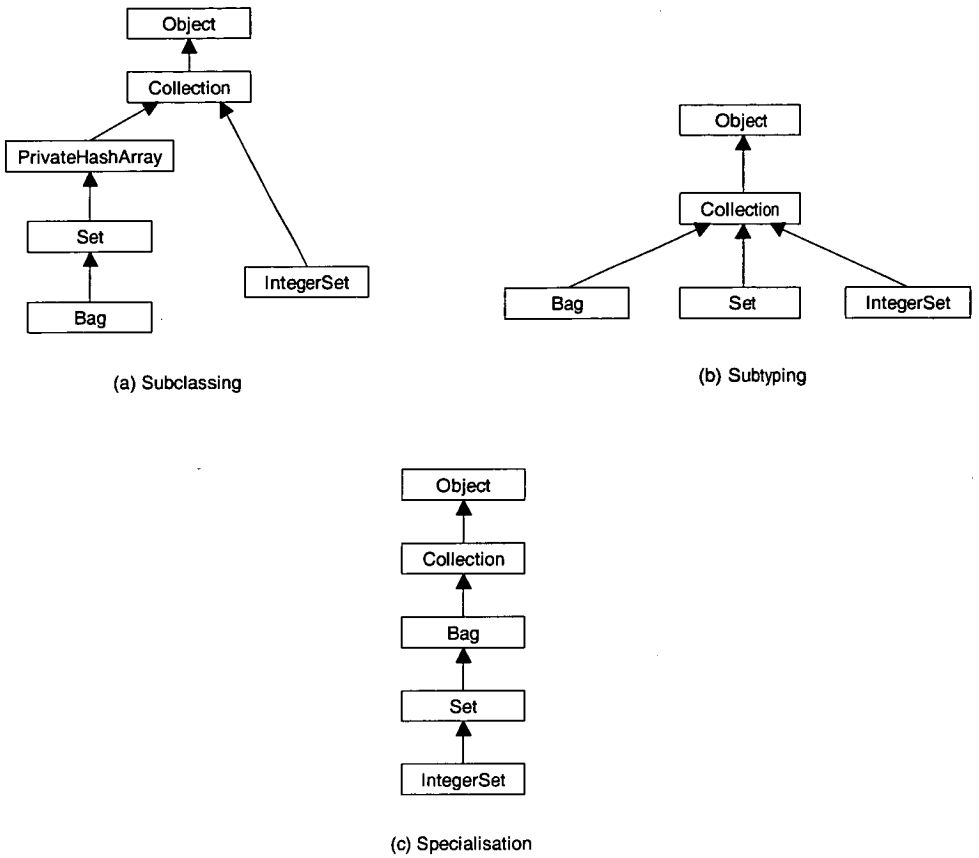


Figure 3.1: The three relationships distinguished [126, p. 58]

As well as aiding the understanding of complex systems, the main benefits of using inheritance for conceptual specialisation include the reuse of code and often the ability to substitute subclasses. Therefore many rewards are gained when incremental change is used to achieve conceptual specialisation. The fundamental question is whether we can guide the developer to such appropriate use of incremental change. We examine existing proposals that constrain incremental change in the next section.

3.2 Constraining Incremental Development

Wegner and Zdonik [201] have identified four different levels of compatibility for incremental modification in object-oriented systems. The weakest of these, *cancellation*, allows the operations of the class to be freely redefined, and even removed from the subclass. The second level, *name compatibility*, allows operations to be redefined, but requires the subclass to preserve the same set of names (that is, no properties may be removed). The third level, *signature compatibility*, requires full syntactic compatibility. The fourth level, *behaviour compatibility*, requires behavioural compatibility between classes and their subclasses.

A common approach in programming languages to constrain incremental change is to impose statically-checkable constraints on the signatures of types to ensure signature compatibility. This approach establishes that all services are provided by the subtype and that no failure will occur upon substitution. However, it says nothing about the behaviour

of the system after substitution. For this reason many consider signature compatibility to be insufficient. A simple example is that a stack could be signature compatible with a queue, but substituting a stack for a queue will lead to incorrect behaviour [132]. (Signature compatibility and the associated compile-time type checking still has advantages such as the elimination of errors that would otherwise be manifest at run-time, and improved reliability, readability and efficiency [140].) This has led researchers to propose subtyping principles based on behaviour compatibility. Subtyping that requires behavioural compatibility is referred to as *behavioural subtyping*. Leavens proposes the following behavioural substitutability principle:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T. [128] as cited by Liskov [131, p. 25].

This principle, sometimes referred to as the *strong substitutability principle*, requires complete behavioural compatibility between the type and subtype. Upon first consideration, constraints on incremental change that ensure strong substitutability would seem to be most desirable, as they ensure both substitutability and conceptual specialisation. However, it has been argued that this is not necessarily the case.

The main concern with constraints requiring substitutability are their applicability in practice. After all, the strong substitutability principle implies that systems cannot detect the difference between the subclass and superclass. Some authors have deemed it necessary to relax the substitutability requirement so that it can be applied in practice. Wegner and Zdonik's comments are particularly pertinent here:

> The requirement of substitutability and the associated notion of subtype and [full] behavioural compatibility is too strong in many practical situations. ... template modification (which is at the heart of subclassing) is more powerful than subtyping as an incremental modification mechanism but also less tractable" [201, p. 65].

This concern for the practical use of constraints requiring strong substitutability has seen various proposals for behavioural subtyping made. Each proposal requires different levels of behavioural compatibility between the type and subtype. Two main approaches have emerged. The first approach, taken by a number of authors [11, 133, 67, 140] involves comparing the methods of the subtype and supertype according to their pre and postconditions, together with the preservation of global properties on the type.

The second approach involves directly comparing the dynamic behaviour (ordering of method execution) of classes, not their methods in isolation. This approach takes the view of classes being *active* entities, and is therefore appropriate for behavioural subtyping relations in concurrent object-oriented languages. In this approach the implementation details of the methods are not important, and so each method is usually represented by an identifier, for example, a labelled transition in a Labelled Transition System (see Definition 2.18). Many of the behavioural subtyping relations focusing on the dynamic behaviour of classes are based on some process algebra correctness relation involving the traces and failures of a class [60, 169, 148, 35, 17]. Other proposals require bisimulation to hold between the classes [7, 17]. We examine several of these proposals in the next section.

3.2.1 Proposals for Behavioural Subtyping

Nierstrasz [148] suggests that substitutability should be checked using the *Request Substitutability* preorder (RS-preorder). The RS-preorder requires that the traces of the refined system include the traces of the original, and if a trace of the original system is performed by the refined system, then its possible failures must be a subset of the corresponding failures of the original. Since failures equivalence is undecidable in general [97], Nierstrasz proposes that services be restricted to those that can be specified by a finite state protocol.

The RS-preorder is based on the notion of active objects providing a service to their environment. A service is determined by the service requests that can be satisfied and hence the name of the preorder. The motivation for this approach is clearly that of substitutability: a server is replaced by another offering the same service. It is not concerned with reuse of specifications. It will not allow the development of a server specification, where the original component can support a sequence of requests, but where an incrementally changed component can only accept that sequence if additional conditions are satisfied, perhaps with additional service parameters.

Bowman et al [35] consider behavioural subtyping in LOTOS [98]. They are dissatisfied with the above proposal of Nierstrasz because they wish to maintain compatible behaviour even when the environment attempts to make a request that is not part of the agreed service. In such a situation, the original server would deadlock, and this behaviour would be expected of a compatible substitute. Such behavioural compatibility is satisfied by *reduction*. One system reduces another if it has a subset of traces and if, after a matching trace, it has a subset of refusals. The problem with this is that it does not allow the refined system to include additional methods, as is common in sub-classing in object-oriented languages. To counter this, they introduce the notion of undefined behaviour which corresponds to calling an undefined method in a sequential system. In the context of LOTOS[1], undefined behaviour can accept or refuse any request. Now, the addition of a method in a refinement will result in defined rather than undefined behaviour (as in the original). Thus, the refinement represents a reduction of the original, thus maintaining the desired properties (identified above). This approach has the strange result of saying that the process *i; a; **stop** [] b; c; **stop*** can be refined by *a; **stop** [] c; **stop*** and by *a; **stop** [] b; a; **stop***. Clearly, the internal action *i* can be taken, thus committing the system to perform *a; **stop***. One of the motivations for the above approach is claimed to be incremental system development, but it is unclear how this is achieved since the only concern seems to be a strongly constrained form of substitutability. The specific example cited is that of a trader in the context of an object oriented platform such as CORBA [150], the TINA DPE (Distributed Processing Environment) [4] or the ODP (Open Distributed Processing) Computational Model [1]. A trader is an object used in order to locate required services. The trader identifies a server that will respond to the desired requests, and as long as this functionality is supported, additional functionality may be present in the chosen server. In this context, it is not clear why Nierstrasz's proposal is inadequate, that is, why the refinement of the server should also reject requests that cannot be handled by the original service specification.

van der Aalst and Basten [7] have used *Labelled Place Transition Petri Nets* to model object life-cycles. Labelled transitions correspond to the methods, and the net structure captures the possible ordering of method calls. It is possible to inherit and incrementally

[1] See Section 2.2 for a brief introduction to LOTOS.

change a life-cycle and van der Aalst and Basten propose different definitions of the relationship that should hold between the life-cycle of a class and that of subclass. In these proposals the refined system can add methods and is required to be branching bisimilar with the original provided that the added methods are either blocked (*protocol inheritance*) or hidden (*projection inheritance*). They also define *life-cycle inheritance*, which is a combination of protocol and projection inheritance.

van der Aalst and Basten [19] recognise that proving bisimulation can be difficult and so present four inheritance-preserving transformation rules that can be used to derive subclasses. These transformations are motivated by the difficulties in proving bisimulation, and there is little evidence that such transformations will be useful in practice.

The implication of the van der Aalst and Basten approach is that the occurrence of additional methods voids any requirements on behaviour. For example, consider the nets N_1 and N_2 of Figure 3.2. The label of each transition in these nets is shown inside the transition. Therefore in the net N_1 the occurrence of the transition with label b can follow the transition with label a, but not vice versa. The net N_2 is an incrementally changed version of N_1 where the newly added components are rendered with darker and thicker lines. N_2 has the same behaviour as N_1, but also allows the possibility of transition sequence cba. That is, N_2 allows the transition with the label b to occur before the transition with the label a. In spite of this N_2 is a valid incremental change from N_1 according to van der Aalst and Basten, since by blocking the transition labelled c in the net N_2 then N_2 is bisimilar to N_1.

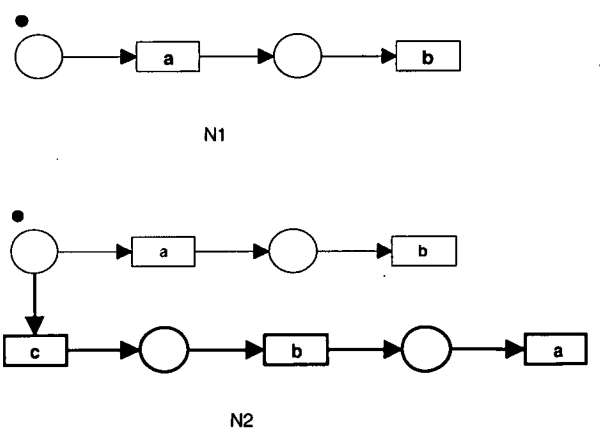


Figure 3.2: A valid incremental change according to van der Aalst and Basten

An example by Wehrheim [202] can be used to illustrate the problem with allowing refinement such as that demonstrated in Figure 3.2. The example, shown in Figure 3.3 (where newly added components are rendered with darker and thicker lines) is that of an automatic teller machine, where the refinement introduces new actions for printing the balance of the account. If the *balance* transition is blocked, then the refined system is bisimilar to the original. However a basic property of the original system, namely entering a pin before withdrawing money, does not hold in the refined system.

Thus, van der Aalst and Basten's definitions imply that newly added methods can change the behaviour (i.e. order of execution) of pre-existing methods. That is, the changed version can exhibit behaviour that is inconsistent with the behaviour of the original. Such change may be appropriate for workflow modelling, but seems too unconstrained for incremental development.

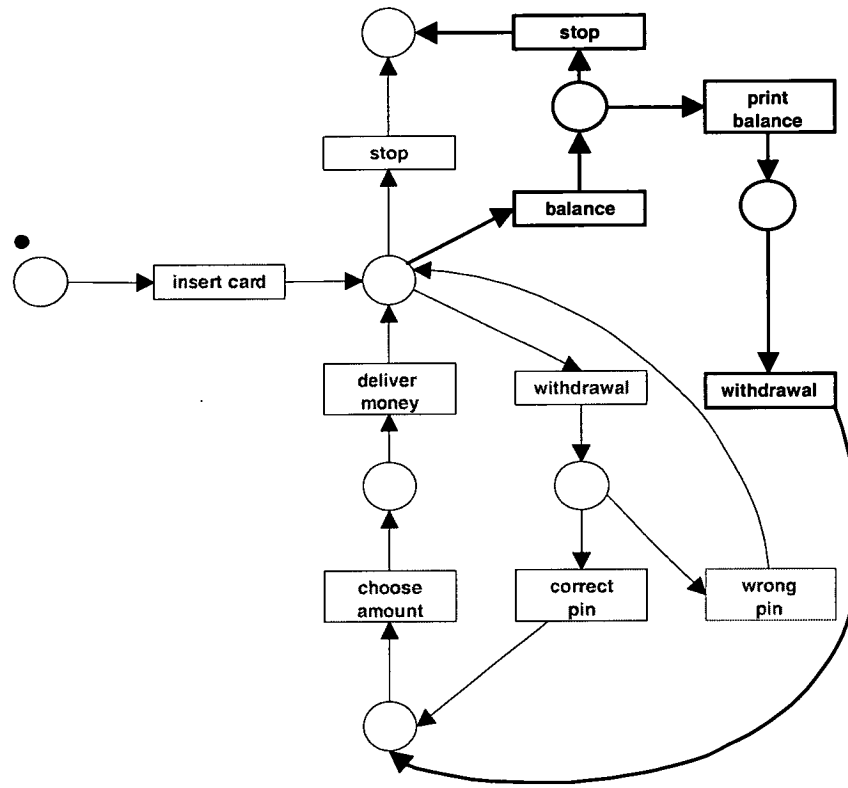


Figure 3.3: ATM example (modified from [202, p. 8])

It appears that if newly added methods are associated with pre-existing methods, then some sort of behavioural consistency should be imposed. Wehrheim [202] proposes three subtype relations and suggests that the most appropriate subtyping relation be chosen depending upon the application domain. Once again, the motivation behind the proposed relations is substitutability: that any changes to the subtype should be transparent to users of the supertype. The first relation proposed suffers from the same problem as van der Aalst and Basten. The next two relations strengthen this first relation so that even if a new service of the subtype is used, no new possibilities of using the old service occur. The second relation ensures that no new possibilities of using the old service occur when only one client accesses the services of the subtype at any one moment. The third and strongest subtyping relation ensures that no new possibilities of using the old service occur when multiple clients can access the service of the subtype at any one moment.

Under the Wehrheim proposals, two systems are related based on their traces and failures together with *restriction* and *concealment*. Restriction renames actions into invisible internal actions (denoted by τ). Concealment is used to ensure that if a new service is used, the possibility of using old services does not arise. It renames actions to invisible but not internal actions (denoted by ν). A problem identified by Wehrheim [202] with the proposals as they stand is that there are no syntactic conditions that help in the checking of the proposed subtype relations. This has been left as a matter for future work.

Balzarotti et al [17] present a range of choices for the semantics of inheritance. They observe that previous proposals for behavioural compatibility — life-cycle inheritance of van der Aalst and Basten and the RS-preorder of Nierstrasz — are often too strong and cannot be applied in many cases. So they generalise these earlier proposals with the possibility of renaming methods. This allows a refinement to support the methods of the original

but under a different name. The strongest preorder, *strong substitutability*, is the protocol inheritance of van der Aalst and Basten (where additional methods are blocked and bisimilarity is required of the result). Two weaker forms of preorder are *strong substitutability with renaming* (that is, the above approach with the possibility of renaming methods) and *weak substitutability* (which is the general form of life-cycle inheritance proposed by van der Aalst and Basten, where added methods can be blocked or hidden). Weaker than both of these is the form of preorder called *weak substitutability with renaming*, which is derived in the obvious way. The above preorders are all based on action observability and are considered appropriate for subtyping.

Balzarotti et al also consider the ST-preorder [160] appropriate for reuse of specifications. The ST-preorder has a more flexible renaming of states (than for actions) since there is an injective map from the Observable Local State Transformation (OLST) algebra [160] of the abstraction into the refinement. In other words, they acknowledge the difference between substitutability and code reuse, and provide different criteria for each.

3.2.2 Constraints on Incremental Change in Practice

Although there have been several proposals that can be used to constrain incremental change in concurrent systems, these proposals are rarely adopted in object-oriented languages. It is difficult to express behavioural constraints in non-formal implementation languages such as C++ [5] and Java [15]. Commonly such non-formal languages require only signature compatibility. An exception to this is Eiffel, which captures the behaviour using a class invariant, and pre and postconditions for each method. The subtype relationship holds if and only if the behaviour of the incrementally changed type satisfies:

- the invariant of the subclass is stronger than that of the superclass
- the precondition of each subclass method is weaker than that of the corresponding superclass method
- the postcondition of each subclass method is stronger than that of the corresponding superclass method.

This approach focuses on the definition of classes and methods as opposed to the dynamic behaviour of the class (that is the ordering of method execution).

The dynamic behaviour of an object is traditionally captured in the form of a finite state machine. For example, the UML uses a form of state transition diagram known as Statecharts (see Appendix A.2) for the specification of dynamic behaviour. Version 1.3 of the UML [6] does not specify any specific policy of Statechart refinement, but instead supports a flexible mechanism that could allow a wide range of policies. Three such policies are discussed: subtyping, strict inheritance, and general refinement. The strongest of these is subtyping, which is intended to constrain the behaviour of the incrementally changed class. Subtyping is based on the rationale that the subtype preserves the pre and postcondition relationships of applying events/operations on the type. The pre and postconditions are realised by the states, and the relationships are realised by the transition. The policy is [6, p2-156]:

- States and transitions are only added, not deleted.

- A refined state has the same outgoing transitions, but may add others, and a different set of incoming transitions. It can have a bigger set of substates, and can change its concurrency property from false to true.
- A refined transition can lead to a new target state that is a substate of the state specified in the base class. This guarantees the postcondition specified by the base class.
- A refined guard has the same guard condition, but can add disjunctions. This guarantees that preconditions are weakened rather than strengthened.
- A refined action sequence contains the same actions (in the same sequence), but can have additional actions. The added actions should not hinder the invariant represented by the target state of the transition.

It is apparent that these constraints are motivated by some kind of implementation of the state machines rather than properties of the state machines themselves. Thus, the concentration on pre and postconditions is appropriate to the definition of classes and methods but not to finite state machine behaviour, where one should consider the occurrence of transition sequences.

As we observed in the previous section, several proposals impose constraints on the traces of method identifiers to ensure appropriate incremental change. However many object-oriented languages that support the specification of dynamic behaviour do not constrain inheritance. For example, the object-oriented Petri Net languages LOOPN++ [115], Esser's OOPN [71], and PN talk [46] have no constraints on the behaviour of an incrementally changed class. These languages only require signature compatibility that can be statically checked. These languages allow transitions to be freely added and redefined in a subclass. None of them support the deletion of existing transitions from a subclass.

The OOPN language CO-OPN/2 [28, 29] supports both subtyping and subclassing. The subclassing is unconstrained and allows net components to be added, deleted or refined. The subtype hierarchy does not have to coincide with the subclass hierarchy, but in the published case studies using CO-OPN/2 the hierarchies have been the same. A class, (possibly defined using subclassing), is strongly substitutable for another class if there is a bisimulation between the second class and the first (restricted to the behaviour of the second). CO-OPN/2 does not provide a guide to the changes that can be made to the class to produce a bisimilar subclass. The bisimilarity is not checked by the CO-OPN/2 tool, but must instead be proved by the user. Proof of a behavioural relation such as bisimulation is difficult. Automatic algorithms to prove such bisimulation are often infeasible in practice. For example, algorithms for bisimulation in one-safe Petri nets are PSPACE-complete [70].

The language CLOWN (CLass Orientation With Nets), [22], uses a finite state machine to describe the life-cycle of each class and an algebraic specification (written in OBJ3 [80]) to describe their abstract data types. Incremental change requires the *ST-preorder* relationship (introduced in [160]) to hold. It is intended that all subclasses in CLOWN satisfy this relationship, but the relationship is not automatically proved by the CLOWN tool. Instead it is left up to the specifier to prove by hand which is relatively easy since the net is a finite state machine. As with CO-OPN/2, CLOWN does not provide a method or guide as to the changes that can be made to the class to produce a behaviourally correct subclass.

Cooperative Nets (and the closely related Cooperative Objects) [21, 177] allow for a refinement of a system by replacing one or more transitions in an existing net, say N_1, by service requests on a newly added subnet, say N_2. The behavioural constraints imposed on this incremental change are that the new net will supply at least the services supplied by N_1 (and possibly more), and that the new net is able to satisfy more requests than N_1. The later constraint seems to run counter to the intuitive notion of conceptual specialisation since the incremental change allows for additional system behaviours.

3.3 Summary

The benefits of incremental change as a mechanism for specialisation have been widely recognised. Various proposals to constrain incremental change in concurrent systems have been made. Typically such proposals have focused on the substitutability of the incrementally changed component. One reason for this is that often a conceptually specialised component is in some sense substitutable for the original component. In fact, in the literature substitutability is commonly said to be equivalent to conceptual specialisation [17, 201, 11]. However, as LaLonde and Pugh [126] argue, there is a significant difference between subtyping and specialisation.

Concerns have been raised that constraints that require substitutability are too strong for use in practice. These concerns are supported both by our own experience, and by the incremental change used in the case studies from the literature that we have examined (see Chapter 5). In these studies, the incrementally changed components are not completely behaviourally compatible with the original components, even though they are valid conceptual specialisations. Further to this, the focus on substitutability has meant the constraints are not statically checkable and that there is no guide to the changes that can be made so that the required conditions hold. We believe it is for these reasons that typically only signature compatibility is adopted in practice.

Requiring signature based compatibility can increase the expressive power of incremental change, but at the same time it decreases structural clarity because so much less can be inferred about the properties of the descendants. What we need, therefore, are constraints that lie between full behaviour compatibility and signature compatibility, where maximum possible expressive power is supported while maintaining conceptual specialisation. Of utmost concern is that they are usable in practice. To achieve this, the constraints should be statically checkable, guide the developer to the changes that can be made, and applicable in practical situations. In Chapter 4 we present a proposal that is statically checkable and guides the developer to the types of change that can be made. In Chapter 5 we examine its applicability in practice.

Chapter 4

Incremental CPN Modelling

In Chapter 3 we observed that many proposals constraining incremental change are focused on substitutability. A variety of constraints have been imposed in order to guarantee substitutability in one form or another. However, such constraints are so strong that it is often difficult to apply incremental change in practice. Rather than requiring substitutability, Lakos argues [116, 125, 124] that incremental change should ensure conceptual specialisation and proposes that for every (complete) refined behaviour there should be a corresponding abstract behaviour, that is: if for each (complete) behaviour of a system S there is a corresponding behaviour in system T, then S is a conceptual specialisation of T.

In this chapter we present Lakos' proposal [116], which we refer to as *Incremental CPN Modelling*. This chapter forms the basis of subsequent chapters and much of it has been previously published [116].

The above general principle is given in terms of morphisms on Coloured Petri Nets [116]. In order to guide the developer to the forms of change that can be used to ensure this principle, Incremental CPN Modelling identifies three specific forms of refinement: *type refinement*, *subnet refinement*, and *node refinement*. Constraints on these refinements ensure that the above principle can be statically checked.

The above principle does not require that an abstract behaviour has a corresponding refined behaviour, and therefore does not necessarily imply complete behavioural compatibility. This in turn means substitutability in the sense Leavens [128] defines it (see Section 3.2) is not guaranteed. We conclude this chapter with an examination of the relationship between this principle and complete behavioural compatibility (bisimilarity).

4.1 A Simple Example

This section presents type, subnet and node refinement using a simple example. The example is taken from the tutorial provided with the HLPN Standard (Draft 3.4) [3]. The system is described as follows [3, p. 34]:

> Two companies, A_Co and B_Co, reside in different cities. A_Co packs and sends big crates of equal size to B_Co, one by one. B_Co has a room where the crates are stored. Crates may be taken from the store-room for processing (e.g. distributed to retails or opened and the contents consumed ...).

In the tutorial this system is modelled by the net of Figure 4.1. In this net a crate can be sent from A_Co (by firing *sendCrate*). The crate will then be in transit (in the place *Crates_in_transit*). A crate in transit can be received (by B_Co by firing *receiveCrate*). It will then be stored (in place *Store_room*) and subsequently processed (by firing *processCrate*). This is an abstract view of the system that can be refined to add more detail.

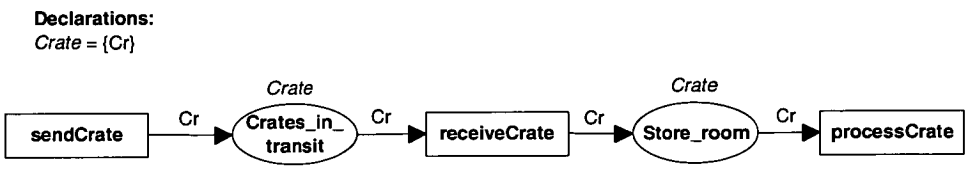


Figure 4.1: Crate sending example [3, p. 34]

The first and simplest form of refinement, *type refinement*, involves incorporating additional information in the tokens and firing modes, while leaving the net structure unchanged. This simply involves extending the token type, and extending the corresponding type for the various transition firing modes. For example, it may be desirable to introduce a further classification of crates to identify the contents of the crates. The refined crate type might be $RefinedCrate = Crate \times String$. That is, each token of the refined net would additionally contain a string describing the contents of the crate. An example of a refined token of this type is $\{Cr, \text{"food"}\}$. This token indicates that the crate contains food. In the refined net the *receiveCrate* transition simply consumes and generates a token of type *RefinedCrate*. Each value of the refined type can be projected onto a value of the abstract type. For example, the refined token $\{Cr, \text{"food"}\}$ projects to the abstract token $\{Cr\}$. By projecting refined values to abstract values it can be seen that if there is a behaviour of the refined system, then there is a corresponding behaviour of the abstract system.

The second form of refinement, called *subnet refinement*, involves augmenting a subnet with additional places, transitions, and arcs. (Also classified as subnet refinement is the extension of a token type or mode type to include extra values that are independent of previous processing. The extension of the type in this way corresponds to the addition of new net components in the unfolded net. Unlike type refinement, where every refined value is projected to an abstract value, under subnet extension of the type, the values of the extended type are not projected onto values of the abstract type, but are ignored in the abstraction. This may mean that some refined tokens have no corresponding abstract token.) In the tutorial, subnet refinement is used to refine the abstract net to model the following behaviour [3, p. 34]:

> The store-room of B_Co can only hold a certain number, say MAX, of these crates. In order to avoid being forced either to leave crates in the street or to rent another store room, B_Co agrees with A_Co on a "flow control protocol".
>
> To implement the protocol, A_Co keeps a record of *Sending Credits*, while B_Co keeps a record of empty "slots" available for placing crates in the store. Any time there are empty slots, B_Co may give the number of empty slots as sending credits for crates to A_Co. B_Co does this by sending a letter with this number and setting the number of empty slots to 0.

The refined net, adapted from the Petri Net Standard tutorial to match our notation for CPN diagrams, is shown in Figure 4.2. The top of this figure is the same as the original net

(Figure 4.1). Places, transitions, and arcs have been added to the original net and these are rendered with thicker and darker lines. Now the *sendCrate* transition can only occur if the number of sending credits is greater than 0. Each time a crate is processed, the number of empty slots is increased. The transitions *sendLetter* and *receiveLetter* model a letter being sent from B_Co and received by A_Co respectively. This new net qualifies as a subnet refinement of the original since ignoring the newly added components in a behaviour of the refined net results in a behaviour of the original net.

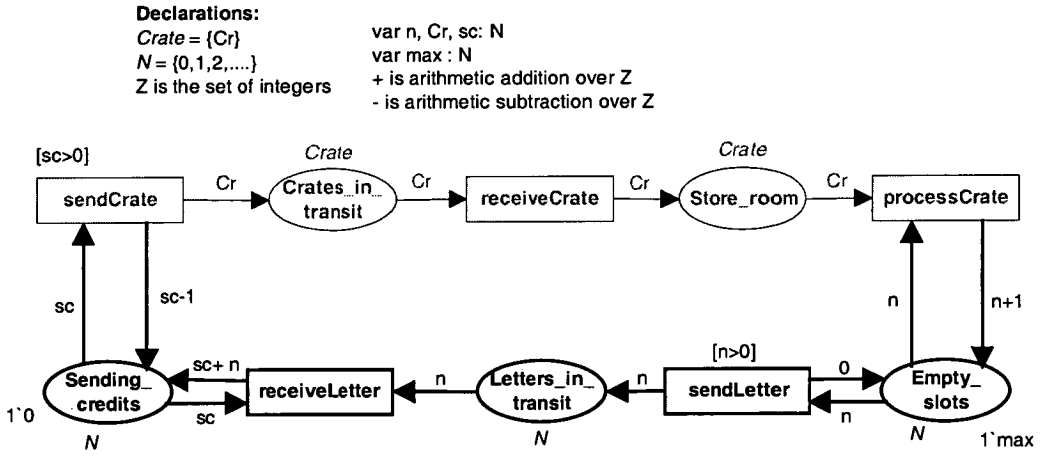


Figure 4.2: The crate sending model refined using subnet refinement [3, p. 35]

The third form of refinement, called *node refinement*, involves replacing a place (transition) by a place (transition) bordered subnet. We refer to the subnet that replaces an abstract place p'' as the *superplace* of p'', and the subnet that replaces an abstract transition t'' as the *supertransition* of t''. A *supernode* is either a superplace or a supertransition. We call the transitions of the refined net that are not part of a supernode *external transitions*. Similarly we refer to the places of the refined net that are not part of a supernode as *external places*. Lakos [116] advocates the use of canonical forms of such refinements. The basis for a canonical superplace is given in Figure 4.3.

It has separate input and output border places, and in this case there are two of each. Each input (output) border place can have more than one incident input (output) arc from (to) the environment. Each input border place has an associated *accept* transition that will transfer tokens from the border place to an internal place, here called *buf*. Similarly, each output border place has an associated *offer* transition that will transfer tokens from the place *buf* to the output border place. All the border places and the place *buf* have the same token type, which is also the mode type shared by the *accept* and *offer* transitions. None of these transitions constrain the flow of tokens. The use of the common arc inscription c guarantees conservation of tokens flowing through the canonical basis. For each superplace we can obtain the corresponding marking in the abstract net, referred to as the *abstract marking* of the superplace. Clearly, the abstract marking of such a canonical place refinement is given by the sum of tokens in the border places and the internal place *buf*. An arbitrary superplace will be of the form of the basis of a canonical refinement (as above) augmented by subnet refinement which extends the *accept* and *offer* transitions.

In our running example, such an incremental change might be the identification of the details of the transit of a crate. In other words, the place *Crates_in_transit*, might be replaced by a subnet that takes into account the delay in transit of a crate, the possibility

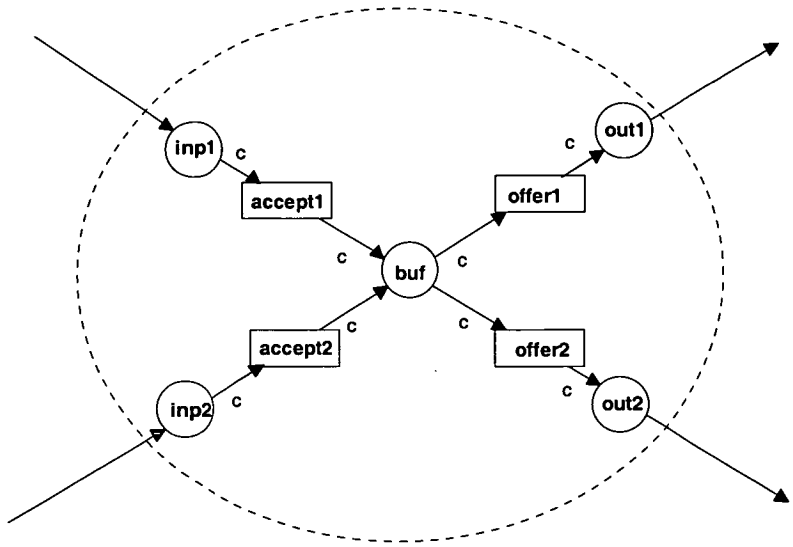


Figure 4.3: Canonical place refinement [116, p. 326]

of damaging crates, etc. Such a subnet (i.e. the *Crates_in_transit* superplace) is shown in Figure 4.4. Note that this has one input border place called *at A_Co* and one output border place called *at B_Co*. The *accept* and *offer* transitions, together with the border places and the internal place *buf* constitute the basis of the canonical superplace. Further activity is achieved by the subnet refinement that extends transitions *accept* and *offer*, where the crate is given to the courier (in place *given_to_courier*) and it can either be damaged (by the occurrence of the *damaged* transition) or it can be successfully delivered (by the occurrence of the *OK* transition). (We note that the duplication of crate tokens in this net does not imply duplication of crates in the physical system since the net is (after all) an abstraction of the system.) This superplace retains the identity of the crates, and hence this information determines the abstract marking of the superplace. Thus, the place *buf* is redundant, since its marking is equivalent to the sum of markings of places *given_to_courier*, *damaged_Crates*, and *for B_Co*. (It is commonly the case that the place *buf* is redundant in superplaces.) Further, this abstract marking is not modified by the various actions internal to the superplace. Clearly, a refined behaviour of the net will have a corresponding abstract behaviour, though the reverse will not necessarily be the case. For example, because an unreliable courier is used then crates may be damaged, and will therefore never arrive at *B_Co*.

For transition refinement, the canonical basis is given in Figure 4.5. It has separate input and output border transitions, in this case there are two of each. Each input (output) border transition can have more than one incident input (output) arc from (to) the environment. Each input border transition has an associated place *recd*, which receives a token equal to the abstract firing mode, when the input border transition has fired with that mode. The transition *switch* can fire when all the input border transitions have fired (with the matching abstract firing mode), thereby completing the input phase. It removes the matching tokens from the *recd* places and puts corresponding tokens into all the *send* places. There is one such *send* place associated with each output border transition. Once such a token is available the output border transition can fire (with the same abstract firing mode). Initially, all the *recd* and *send* places are empty. The abstract firing of the super-transition commences with the firing of any of the input border transitions, and is completed when all the matching output border transitions have fired and the *recd* and *send*

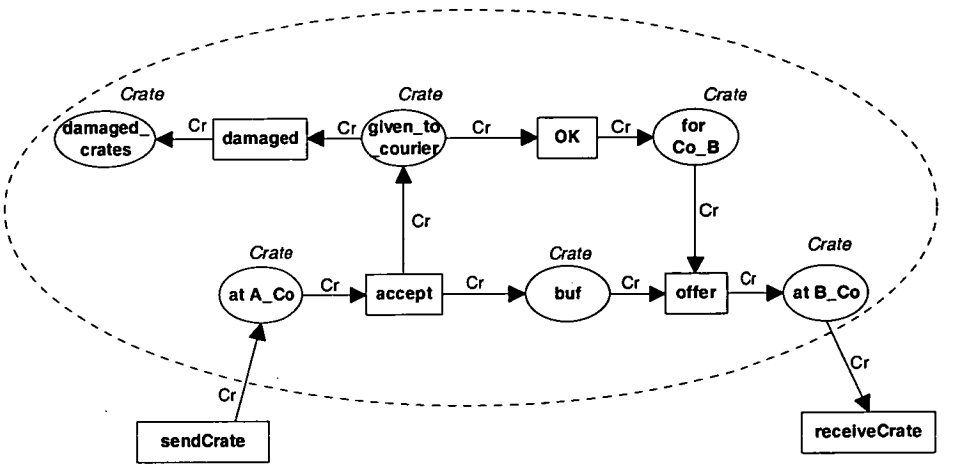


Figure 4.4: Subnet indicating the transit of crates

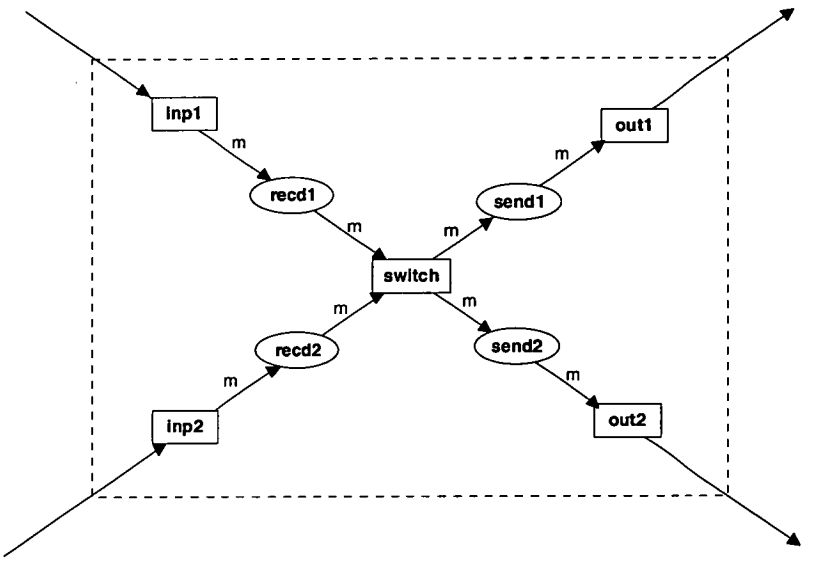


Figure 4.5: Canonical transition refinement [116, p. 328]

places are again empty. Only such completed firing sequences will have corresponding abstract firing sequences. The canonical construction ensures that input border transitions fire before the corresponding output border transitions, thus guaranteeing the enabling of the corresponding abstract transition.

An arbitrary supertransition will be of the form of a canonical refinement (as above) together with a subnet refinement that augments the border transitions and possibly even the *switch* transition. In our running example, we may wish to refine the *processCrate* transition to reflect the component activities that occur when a crate is processed. This might involve unpacking the crate and storing its contents, and at the same time recording that the crate has been delivered. This can be modelled using the supertransition of Figure 4.6.

The transition *Store_crate_contents* may fail to fire (e.g. if the contents have been damaged in transit). In this case, the abstract firing of the supertransition will never complete, and hence such an incomplete refined activity will have no corresponding abstract activity.

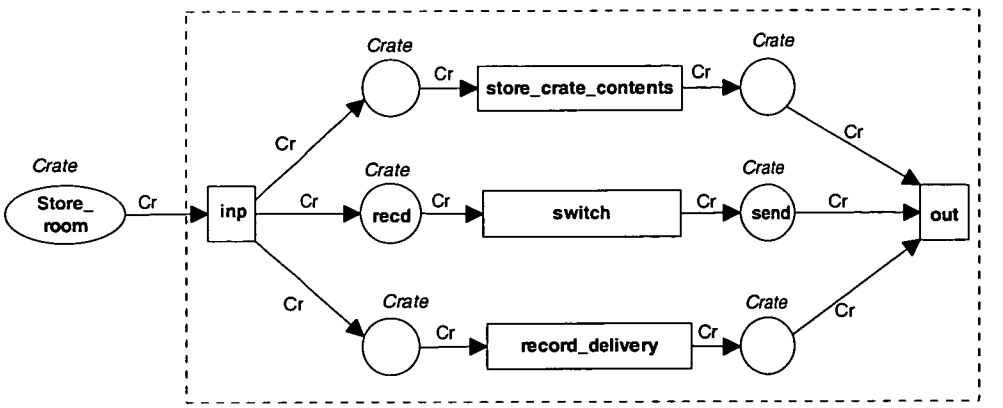


Figure 4.6: Refined processCrate transition

Even though the above three forms 'of refinement can be clearly identified and analysed in isolation, they will commonly be used in combination in practical applications.

4.2 Formalising Refinements

As Desel [63] points out, net transformations play an important role in the analysis and synthesis of systems modelled by nets. Transformations that allow abstraction or refinement are particularly important since they allow hierarchical models to be built, each model with a different level of detail.

A transformation between nets is referred to as a *net morphism.* C.A. Petri describes the notion of a *net morphism* as a function "from one net into another which respects connectivity and orientation" [159, p.3]. Under this definition, net morphisms specify only structural constraints on nets (as opposed to behavioural constraints). That is they depend on, or respect, the structure of nets [77] and are usually graph theoretic in spirit [36]. We will refer to such morphisms as *structure preserving net morphisms* to avoid confusion with the more recent net morphisms that preserve net behaviour.

Reisig [163] has defined abstraction and refinement in terms of structure respecting net morphisms. Here a net N_2 is a refinement of a net N_1 if the morphism is surjective and all arcs in N_1 have a preimage in N_2. Fehling [72] uses structure respecting morphisms to define hierarchical petri nets with building blocks. (The building blocks are slightly restricted hierarchical Petri Nets with hierarchically structured interfaces).

Some ideas of preserving behaviour in graph theoretic low level net morphisms are presented by Desel and Merceron [64, 63]. Winskel [203] provides a definition for morphisms of low level nets that respect behaviour. Here petri nets are viewed as certain 2-sorted algebras and net morphisms are homomorphisms of the corresponding algebras. Another class of behaviour preserving morphisms are defined in a category theory framework by Nielsen et al [147].

Morphisms for high-level nets that preserve behaviour are considered by Padberg et al [151]. Again the framework is category theory. Firstly, morphisms that preserve transitions are defined. Under transition preserving morphisms no new arcs are added to mapped

transitions and no mapped arcs are deleted from their pre and post domains. Transition preserving morphisms are used as the theoretical basis of vertical structuring of high-level nets. Secondly, morphisms that preserve places are defined. Given a place preserving morphism mapping from net N_1 to net N_2 the pre and post domain of a transition in N_2 may contain more places than the original transition in N_1. Place preserving morphisms are used to preserve safety properties (expressed via temporal logic formulas on markings) between nets.

In Section 4.3 we present definitions for CPN morphisms. The notion of *system morphisms* is used to capture behavioural compatibility [116], in contrast to structure respecting morphisms. Type, subnet, and node refinement are defined in Sections 4.4, 4.5, and 4.6 respectively.

4.3 CPN Morphisms

Definition 4.1. [116, def. 3.7]
Given a refined net N and an abstract net N', related by a system morphism $\phi : N \rightarrow N'$ then the *preimage* of $x' \in P' \cup T'$ is $N_{x'} = \phi^{-1}(x')$. We write:

$$N_{x'} = (P_{x'}, T_{x'}, A_{x'}, C_{x'}, E_{x'}, \mathbb{M}_{x'}, \mathbb{Y}_{x'}, M_{0x'})$$

In the remainder of the thesis we use $P_{x'}$ to refer to the set of places of the preimage of $x' \in P' \cup T'$ and $T_{x'}$ to refer to the set of transitions of the preimage of x'.

We denote by P'' the set of places of the abstract net that are replaced by non-trivial subnets in the refined net. Similarly we denote by T'' the set of transitions of the abstract net that are replaced by non-trivial subnets in the refined net.

Definition 4.2. Given a refined net N, and an abstract net N', related by a system morphism $\phi : N \rightarrow N'$ then we define:

a. $P'' = \left\{ p' \in P' \;\middle|\; T_{p'} \neq \emptyset \right\}$

b. $T'' = \left\{ t' \in T' \;\middle|\; P_{t'} \neq \emptyset \right\}$

c. $X'' = P'' \cup T''$.

Note: The set P'' contains the abstract places that are refined by a subnet that includes transitions. The set T'' contains the abstract transitions that are refined by a subnet that includes places.

We refer to the subnet $N_{p''}$ that replaces the abstract place $p'' \in P''$ in the refined net as the *superplace* of p'', and the subnet $N_{t''}$ that replaces the abstract transition $t'' \in T''$ as the *supertransition* of t''. A *supernode* of $x'' \in X''$ is either a superplace or supertransition.

Definition 4.3. For a refined net N, abstract net N', and an abstract node $x'' \in X''$ we define:

a. $inpbdr(N_{x''}) = \{x \in P_{x''} \cup T_{x''} \mid \exists\, y \in P \cup T \setminus P_{x''} \cup T_{x''} : y \in {}^\bullet x\}$

b. $outbdr(N_{x''}) = \{x \in P_{x''} \cup T_{x''} \mid \exists\, y \in P \cup T \setminus P_{x''} \cup T_{x''} : y \in x^\bullet\}$

c. $bdr(N_{x''}) = inpbdr(N_{x''}) \cup outbdr(x'')$

d. $inpenv(N_{x''}) = \{y \in (P \cup T) \setminus (P_{x''} \cup T_{x''}) \mid \exists\, x \in P_{x''} \cup T_{x''} : y \in {}^\bullet x\}$

e. $outenv(N_{x''}) = \{y \in P \cup T \setminus P_{x''} \cup T_{x''} \mid \exists\, x \in P_{x''} \cup T_{x''} : y \in x^\bullet\}$

f. $env(N_{x''}) = inpenv(N_{x''}) \cup outenv(N_{x''})$

Note:

a. $inpbdr(N_{x''})$ denotes *input border* of the supernode of x'', that is, the set of nodes of the supernode of x'' that have input from the environment of the supernode of x''.

b. $outbdr(N_{x''})$ denotes *output border* of the supernode of x'', that is, the set of nodes of the supernode of x'' that have output to the environment of the supernode of x''.

c. $bdr(N_{x''})$ denotes the *border* of the supernode of x''.

d. $inpenv(N_{x''})$ denotes the *input environment* of the supernode of x'', that is, the set of nodes of the supernode of x'' that are input to the supernode of x''.

e. $outenv(N_{x''})$ denotes the *output environment* of the supernode of x'', that is, the set of nodes of the supernode of x'' that are output of the supernode of x''.

f. $env(N_{x''})$ denotes the *environment* of the supernode of x''.

Definition 4.4. [116, def. 3.8]
A *net morphism* $\phi : N \to N'$ is a mapping from N to N' that is *structure-respecting*, namely:

a. ϕ is surjective with respect to P', T', A'

b. $\forall (x,y) \in A \cap P \times T : \phi(x) = \phi(y) \ \vee \ (\phi(x), \phi(y)) \in A' \cap (P' \times T')$

c. $\forall (x,y) \in A \cap T \times P : \phi(x) = \phi(y) \ \vee \ (\phi(x), \phi(y)) \in A' \cap (T' \times P')$

This is the common definition of net morphisms (albeit with various additional constraints) which is primarily concerned with respecting the adjacency properties of the net [23, 64, 65, 72, 159, 163]. It does not constrain behaviour (except indirectly) — in fact, sets of markings and steps are not normally included. It also does not encompass restriction on places and transitions, that is, where selected places and transitions (and their associated arcs) are ignored by the morphism.

Proposition 4.5. Given two net morphisms $\phi_1 : N \to N'$ and $\phi_2 : N' \to N''$ their composition $\phi = \phi_2 \circ \phi_1 : N \to N''$ is a net morphism.

Proof. See Lakos [116, prop. 3.9]. ◊

Just as it is common to distinguish nets and net systems, we now define *system morphisms* which respect behaviour and not just structure. In order to define the behaviour respecting properties of a system morphism it is desirable to consider complete steps, since the firing of multiple transitions in the refinement can correspond to the firing of one transition in the abstraction.

Definition 4.6. [116, def. 3.10] as modified in [120]
Given a morphism $\phi : N \rightarrow N'$, a step Y of N is *complete* if $\forall t' \in T' : \forall t \in bdr(N_{t'})$:

$$Y(t) = \phi(Y)(t')$$

Thus a step is complete if all border transitions occur with matching modes (that also match the mode occurrence of the corresponding abstract transitions). We refer to a step sequence $Y^* \in \sigma\mathbb{Y}$ as a complete firing sequence if it is the realisation of a complete step.

Definition 4.7. [116, def. 3.11] as modified in [119]
A *system morphism* $\phi : N \rightarrow N'$ is a mapping from N to N' that is *behaviour-respecting*, namely:

a. ϕ is surjective with respect to P', T', A'

b. ϕ is linear and total over both $\mathbb{M}$ and $\mathbb{Y}$

c. $\forall M \in \mathbb{M}_R : \forall Y \in \mathbb{Y}$:
Y is realisable as $Y_1 Y_2 \ldots Y_n$ at marking $M \Rightarrow$
$\phi(Y)$ is realisable as $\phi(Y_1)\phi(Y_2)\ldots\phi(Y_n)$ at marking $\phi(M)$.

d. $\forall M \in \mathbb{M}_R : \forall Y \in \mathbb{Y} : Y$ is complete $\Rightarrow$
$\phi(M + E^+(Y) - E^-(Y)) = \phi(M) + \phi(E^+)(\phi(Y)) - \phi(E^-)(\phi(Y))$

Note:

c. If the refined step is realisable then its realisation can be used to derive the realisation of the corresponding abstract step, by projecting or restricting each component step.

d. This modified rule for the effect of a refined step (cf. the definitions of Winskel [203]) is used since we cannot consider the component steps (of its realisation) in isolation. Thus, part (c) guarantees the enabling of the abstract sequence, while part (d) captures its overall effect.

This definition captures the requirement identified earlier, namely that refinement constrains the behaviour of the system since refined behaviours have a corresponding abstract behaviour. (Note that examples of system morphisms are given with the more specific definitions in the following sections.)

Proposition 4.8. Given two system morphisms $\phi_1 : N \rightarrow N'$ and $\phi_2 : N' \rightarrow N''$ their composition $\phi = \phi_2 \circ \phi_1 : N \rightarrow N''$ is a system morphism.

Proof. See Lakos [116, prop. 3.12]. ◊

The above proposition tells us that we can combine the different forms of refinement (considered in subsequent sections) and we still have a system morphism.

4.4 Type Refinement

The first and perhaps simplest form of refinement is to retain the structure of the net without modification, but to replace some (or all) of the token and mode types by subtypes. Given the formulation of CPNs, where the arc inscriptions are functions (from modes to token multisets), it may not even be necessary to change the arc inscriptions, provided that they are given by polymorphic functions. For example, if an arc inscription is the identity function (that is, the mode determines the token to be added or removed), then a simple change to both mode and token type would give a refined behaviour. If the arc inscription functions are *not* given by polymorphic functions, then they will need to be replaced, but the new versions must be consistent with the old. The distinctive thing about type refinement is that there is a projection function from subtype to supertype so that every refined state or action has a corresponding abstract state or action.

Definition 4.9. [116, def. 3.13]
A morphism $\phi : N \rightarrow N'$ *captures a type refinement* if:

a. ϕ is an identity function on P, T, A, i.e. $\forall p \in P : \phi(p) = p$, etc.

b. $\forall x \in P \cup T : C(x) <: \phi(C)(x)$

c. $\forall x \in P \cup T : \forall c \in C(x) : \phi(1`(x,c)) = 1`(x, \Pi_{\phi(C)(x)}(c))$

d. $\forall (p,t) \in A : \forall (t,c) \in \mathbb{Y} :$
$\phi(E^-(1`(t,c)))(p) = \Pi_{\phi(C)(p)}(E(p,t)(c)) = \phi(E)(p,t)(\Pi_{\phi(C)(t)}(c))$
$\forall (p,t) \in A : \forall (t,c) \in \mathbb{Y} :$
$\phi(E^+(1`(t,c)))(p) = \Pi_{\phi(C)(p)}(E(t,p)(c)) = \phi(E)(t,p)(\Pi_{\phi(C)(t)}(c))$

Note:

a. There is no change to the structure of the net (to places, transitions and arcs).
b. The colours associated with the places and transitions are consistently subtyped.
c. Given the consistent structure, the morphism is defined for all markings and steps using the appropriate projection functions (see Section 4.3).
d. The arc inscriptions are consistent, that is, the projected effect of a mode or step is the same as the effect of a projected mode or step.

In Section 4.1 we gave an example where the crate token type was refined to include information about the contents of the crate. There were corresponding changes to the relevant firing modes. For this to be considered a type refinement, we require that the (implicitly defined) morphism, ϕ from the refined net to the abstract net satisfies Definition 4.9. Since there was no change to places, transitions, and arcs, then the morphism clearly satisifies part (a) of Definition 4.9. As required by part (b) the colour of each place or transition of the refined net in the example is a subtype of the type of the place or transition in the abstract net (in the example, the subtype is *RefinedCrate*). Hence part (b) is satisified. Further the morphism is defined for all markings and steps, as required for part (c). Since storing the contents of the crate with each token does not affect the enabling of the transitions then the projected effect of each mode or step is the same as the effect of the projected mode or step, as required by part (d). Hence the first example of Section 4.1 qualifies as a type refinement.

We note that type refinement can eliminate some abstract behaviour since the refined token requirements of a transition may not be satisfied, even though the abstract requirements are. As an example, the firing of *sendCrate* transition may be constrained in the refined net by the contents of the crate, so that any crate with dangerous goods are not be sent.

Proposition 4.10. A morphism $\phi : N \to N'$ that captures a type refinement is a system morphism (in the sense of Definition 4.7).

Proof. See Lakos [116, prop. 3.14]. ◊

4.5 Subnet Refinement or Extension

The second form of refinement is to add net components: places, transitions and arcs, or even additional token or mode values. As a morphism (from refined to abstract nets), this would be called a restriction, since the net components are being discarded or ignored. Where token or mode types are extended, then in contrast to type refinement, abstraction does *not* project the additional refined values onto abstract values but rather ignores them. (In the equivalent unfolded PTN, this is the same as ignoring places, transitions, and arcs.) This does not satisfy the structure respecting requirements for a net morphism (Definition 4.4), but it does qualify as a system morphism (Definition 4.7).

Definition 4.11. [116, def. 3.15] as modified in [120]
A morphism $\phi : N \to N'$ *captures a subnet refinement* if:

a. The net structure is restricted, i.e. $\forall p \in P : \phi(p)$ is defined $\Rightarrow \phi(p) = p$ and similarly for T, A.

b. $\forall x \in \phi(P) \cup \phi(T) : C(x) \supseteq \phi(C)(x)$

c. $\forall x \in P \cup T : \forall c \in C(x) : \phi(1`(x,c)) = 1`(x,c)$ if $x \in \phi(P) \cup \phi(T)$ and $c \in \phi(C(x))$, otherwise $\emptyset$.

d. $\forall Y \in \mathbb{Y} : \phi(E^+(Y)) = \phi(E^+)(\phi(Y))$ and $\phi(E^-(Y)) = \phi(E^-)(\phi(Y))$

Note:

a. The sets of places, transitions, and arcs, may be restricted by ϕ.
b. The colours associated with retained places and transitions may be restricted.
c. Given the consistent colouring, the morphism is defined for all markings and steps.
d. The restricted incremental effect of the step is the same as the incremental effect of the restricted step. This implies that ignored transitions cannot modify retained places.

In Section 4.1 we gave an example of subnet refinement, where the net was augmented to model the limited capacity of the store room. For this to be considered a subnet refinement, we require that the (implicitly defined) morphism, ϕ, from the refined net to the abstract net satisfies Definition 4.11. The places, transitions, and arcs added in the refinement are ignored by the morphism. The other places and transitions map to similarly named abstract places and abstract transitions. Therefore part (a) is satisfied. The type of each refined place that maps to an abstract place is the same as that of the abstract place, and similarly the type of each refined transition that maps to an abstract transition is the same as that of the abstract transition. Therefore part (b) is satisfied. Further the morphism is defined for markings and steps, as required for part (c). Since if a crate can be shipped when store room capacity is considered, then it can be shipped when capacity is not considered, then the projected effect of each mode or step is the same as the effect of the projected mode or step. Therefore part (d) is satisfied. Hence the second example of Section 4.1 qualifies as a subnet refinement.

Proposition 4.12. A morphism $\phi : N \to N'$ that captures a subnet refinement is a system morphism (in the sense of Definition 4.7).

Proof. See Lakos [116, prop. 3.16]. ◊

4.6 Node Refinement

The third form of refinement is the replacement of a place (transition) by a place (transition) bordered subnet. This is referred to as *node refinement* to distinguish it from other forms of refinement being considered, even though traditional Petri Net theory simply refers to it as refinement [23]. Here we present canonical forms of node refinement.

Definition 4.13. [116, def. 3.17]
Given a colour-respecting morphism $\phi : N \to N'$, then $N_{p'}$ is a *canonical place refinement* of $p' \in P'$ provided:

a. $\forall p \in P - P_{p'} : \forall t \in T - T_{p'} :$
$\phi(p) = p \ \wedge \ \phi(t) = t \ \wedge$
$(p,t) \in A \Rightarrow ((p,t) \in A' \ \wedge \ E(p,t) = E'(p,t)) \ \wedge$
$(t,p) \in A \Rightarrow ((t,p) \in A' \ \wedge \ E(t,p) = E'(t,p))$

b. $\forall t \in \text{env}(N_{p'}) :$

$$E'(p',t) = \sum_{(p,t) \in A \cap (P_{p'} \times T)} E(p,t) \quad \text{and}$$

$$E'(t,p') = \sum_{(t,p) \in A \cap (T \times P_{p'})} E(t,p)$$

c. $bdr(N_{p'}) = \{\text{inp1}, \text{inp2}, \ldots \text{out1}, \text{out2}, \ldots\}$

d. $P_{p'} = bdr(N_{p'}) \cup \{\text{buf}\} \cup P_{other}$

e. $T_{p'} = \{\text{accept1}, \text{accept2}, \ldots \text{offer1}, \text{offer2}, \ldots\} \cup T_{other}$

f. $A_{p'} = \{(\text{inp1},\text{accept1}), \ldots (\text{accept1},\text{buf}), \ldots (\text{buf},\text{offer1}), \ldots (\text{offer1},\text{out1}), \ldots\} \cup A_{other}$

g. $^\bullet\text{inp1} \subseteq env(N_{p'}) \ \wedge \ \text{inp1}^\bullet = \{\text{accept1}\} \ \wedge \ \ldots$
$\text{out1}^\bullet \subseteq env(N_{p'}) \ \wedge \ ^\bullet\text{out1} = \{\text{offer1}\} \ \wedge \ \ldots$
$^\bullet\text{buf} = \{\text{accept1}, \ldots\} \ \wedge \ \text{buf}^\bullet = \{\text{offer1}, \ldots\}$

h. $\forall x \in (P_{p'} - P_{other}) \cup (T_{p'} - T_{other}) : C_{p'}(x) = C(x) = C'(p')$

i. $\forall a \in A_{p'} - A_{other} : E_{p'}(a) = \text{Id}$

j.

$$\forall (p,c) \in TE : \phi(1`(p,c)) = \begin{cases} 1`(p',c) & \text{if } p \in P_{p'} - P_{other} \\ \emptyset & \text{if } p \in P_{other} \\ 1`(p,c) & \text{otherwise} \end{cases}$$

$$\forall (t,c) \in FE : \phi(1`(t,c)) = 1`(t,c) \text{ if } t \notin T_{p'} \text{ otherwise } \emptyset$$

Note:

a. Apart from the subnet $N_{p'}$ and its incident arcs, the rest of the net is unchanged.
b. The flow of tokens across the boundary of the place refinement is consistent.
c. The border of the subnet consists of the places inp1,...,out1,....
d-f. The component places, transitions, and arcs consist of the basis places, transitions, and arcs respectively, plus others that constitute the subnet refinement of the accept and offer transitions (cf. Figure 4.3).
g. The input (output) border places only have input (output) from environment transitions, other arcs incident on the basis places are exactly those of part (e).
h. The colour of the basis places and transitions are all the same.
i. The arc inscriptions for all the basis arcs are the same: the identity function.
j. In abstracting the place refinement, the abstract marking is given by the marking of the basis places and the internal actions are ignored.

Definition 4.14. [116, def. 3.19] modified as in [118]
Given a colour-respecting morphism $\phi : N \rightarrow N'$, then $N_{t'}$ is a *canonical transition refinement* of $t' \in T'$ provided:

a. $\forall p \in P - P_{t'} : \forall t \in T - T_{t'} :$
$\phi(p) = p \ \wedge \ \phi(t) = t \ \wedge$
$(p,t) \in A \Rightarrow ((p,t) \in A' \ \wedge \ E(p,t) = E'(p,t)) \ \wedge$
$(t,p) \in A \Rightarrow ((t,p) \in A' \ \wedge \ E(t,p) = E'(t,p))$

b. $\forall p \in \text{env}(N_{t'}) :$

$$E'(p,t') = \sum_{(p,t) \in A \cap (P \times T_{t'})} E(p,t) \quad \text{and}$$

$$E'(t',p) = \sum_{(t,p) \in A \cap (T_{t'} \times P)} E(t,p)$$

c. $bdr(N_{t'}) = \{\text{inp1}, \text{inp2}, \ldots \text{out1}, \text{out2}, \ldots\}$

d. $P_{t'} = \{\text{recd1}, \ldots, \text{send1}, \ldots\} \cup P_{other}$

e. $T_{t'} = bdr(N_{t'}) \cup \{\text{switch}\} \cup T_{other}$

f. $A_{t'} = \{(\text{inp1},\text{recd1}), \ldots, (\text{recd1},\text{switch}), \ldots (\text{switch},\text{send1}), \ldots (\text{send1},\text{out1}), \ldots\} \cup A_{other}$

g. $^\bullet\text{inp1} \subseteq env(N_{t'}) \cup P_{other} \ \wedge \ ^\bullet\text{recd1} = \{\text{inp1}\} \ \wedge \ \text{recd1}^\bullet = \{\text{switch}\} \ \wedge \ \ldots$
$\text{out1}^\bullet \subseteq env((N_{t'}) \cup P_{other} \ \wedge \ \text{send1}^\bullet = \{\text{out1}\} \ \wedge \ ^\bullet\text{send1} = \{\text{switch}\} \ \wedge \ \ldots$
$^\bullet\text{switch} \supseteq \{\text{recd1}, \ldots\} \ \wedge \ \text{switch}^\bullet \supseteq \{\text{send1}, \ldots\}$

h. $\forall x \in (P_{t'} - P_{other}) \cup (T_{t'} - T_{other}) : C_{t'}(x) = C(x) = C'(t')$

i. $\forall a \in A_{t'} - A_{other} : E_{t'}(a) = \text{Id}$

j. $\forall p \in P_{t'} - P_{other} : M_{0t'}(p) = \emptyset$

k.

$$\forall (p,c) \in TE : \phi(1`(p,c)) = 1`(p,c) \text{ if } p \in P - P_{t'} \text{ otherwise } \emptyset$$

$$\forall (t,c) \in FE : \phi(1`(t,c)) = \begin{cases} 1`(t',c) & \text{if } t = \text{switch} \\ \emptyset & \text{if } t \in T_{t'} - \{\text{switch}\} \\ 1`(t,c) & \text{otherwise} \end{cases}$$

Note:

a. Apart from the subnet $N_{t'}$ and its incident arcs, the rest of the net is unchanged.

b. The flow of tokens across the boundary of the place refinement is consistent.

c. The border of the subnet consists of the transitions inp1,...,out1, ...

d-f. The component places, transitions, and arcs consist of the basis places, transitions, and arcs respectively, plus others that constitute the subnet refinement of the border transitions (cf. Figure 4.5).

g. The input (output) border transitions only have input (output) from environment places or the other internal places, the basis places only have the incident arcs of part (e).

h. The colour of the basis places and transitions are all the same.

i. The arc inscriptions for all the basis arcs is the identity function.

j. The initial marking of all basis places is empty.

k. The internal marking of the supertransition is ignored by the abstraction and the firing of the switch transition corresponds to the abstract firing of t'.

In Section 4.1, Figure 4.4 we gave an example where a place was refined to indicate the delay in transit of a crate, the possibility of damaging crates, etc. According to Definition 4.13, the canonical basis of this refinement consists of the input border place (*at A_Co*), the output border place (*at B_Co*), the buffer place (*buf*), and the *accept* and *offer* transitions and corresponding arcs. In the example, subnet refinement augments the *accept* and *offer* transitions. The subnet refinement consists the places, transitions, and arcs that do not form part of the canonical basis. That is, it consists of the places *damaged_crates*, *given_to_courier*, *for_Co_B*, and the transitions *damaged*, and *OK*, and the corresponding arcs.

Similarly, in Figure 4.6 we gave an example of transition refinement to indicate the activity when a crate is processed. According to Definition 4.14 the canonical basis of this refinement consists of the input transition, *inp*, the *switch* transition, and the output transition, *out*, together with the places *recd* and *send* and corresponding arcs. In the example, subnet refinement augments the border transitions. The subnet refinement consists the places, transitions, and arcs that do not form part of the canonical basis. That is, it consists of the transitions *store_crate_contents* and *record_delivery* together with their input and output places and corresponding arcs.

Proposition 4.15. A morphism $\phi : N \rightarrow N'$ with a canonical place refinement constitutes a system morphism (in the sense of Definition 4.7).

Proof. See Lakos [116, prop. 3.18]. ◊

Proposition 4.16. A morphism $\phi : N \rightarrow N'$ with a canonical transition refinement constitutes a system morphism (in the sense of Definition 4.7).

Proof. See Lakos [116, prop. 3.20]. ◊

4.7 Relationship with Equivalences (cf Section 2.3, fig 2.7)

It is helpful to consider the relation of the above refinement proposals to bisimilarity. By doing so we detail what extra is required for system morphisms to be bisimulations, and hence what is required for Incremental CPN Modelling to ensure strong substitutability. The first version of the following was proposed by Lakos [117]. It has since undergone several significant revisions by both the current author and Lakos.

Bisimulation is usually defined in terms of Labelled Transition Systems (see Section 2.3). Clearly by attaching a label to each transition, equivalent definitions can be formed for elementary net systems. Such elementary net systems are referred to as *Labelled Elementary Net Systems* (LENs). There are several possible definitions of bisimulation for LENs. Pomello [160] gives definitions of bisimulation based on interleaving, partial order, and step semantics for LENs. Here we are concerned with Coloured Petri Nets rather than LENs. As defined in Definition 4.19, we consider two CPNs to be bisimilar according to step semantics if their equivalent unfolded nets are bisimilar according to step semantics. Recall from Chapter 2 that Λ is the set of observable actions. $\tau \notin \Lambda$ denotes a special unobservable action and we use Λ_τ to denote $\Lambda \cup \{\tau\}$.

Definition 4.17. Given a net $N = (P,T,A,C,E,\mathbb{M},\mathbb{Y},M_0,L,\Lambda_\tau)$, the labelling function $L: FE \to \Lambda_\tau$ attaches an action name to each firing element. This function can be extended to label steps, $L: \mathbb{Y} \to \mu\Lambda_\tau$, in the obvious way:

$$L(Y) = \begin{cases} \tau & \text{, if } \forall (t,c) \in Y \;\; L((t,c)) = \tau \\ \sum\limits_{a \in \Lambda} |L^{-1}(a) \cap Y| \grave{} a & \text{,otherwise} \end{cases}$$

The function can again be extended, this time so that it labels sequences of steps, $L: \sigma\mathbb{Y} \to \sigma(\mu\Lambda_\tau)$:

$$\forall Y_1 \in \mathbb{Y}, \forall Y_2{}^* \in \sigma\mathbb{Y} \quad L(Y_1Y_2{}^*) = \begin{cases} L(Y_2{}^*), \text{ if } L(Y_1) = \tau \\ L(Y_1)L(Y_2{}^*), \text{ otherwise} \end{cases}$$

Definition 4.18. Let $(P,T,A,C,E,\mathbb{M},\mathbb{Y},M_0,L,\Lambda_\tau)$ be a labelled CPN, where $L: \sigma\mathbb{Y} \to \sigma(\mu\Lambda)$. Let $z^* \in \sigma(\mu\Lambda)$ and $M_1, M_2 \in \mathbb{M}$, then:

a. z^* is enabled at M_1, denoted $M_1 \xrightarrow{z^*}$, iff there exists $Y^* \in \sigma\mathbb{Y}$ such that $L(Y^*) = z^*$ and $M_1[Y^*\rangle$

b. if z^* is enabled at M_1, then the occurrence of z^* can lead from M_1 to M_2, denoted $M_1 \xrightarrow{z^*} M_2$, iff there exits $Y^* \in \sigma\mathbb{Y}$ such that $L(Y^*) = z^*$ and $M_1[Y^*\rangle M_2$.

Now we define (weak) bisimulation and bisimilarity for labelled CPNs.

Definition 4.19. Given two labelled CPNs
$N_1 = (P_1,T_1,A_1,C_1,E_1,\mathbb{M}_1,\mathbb{Y}_1,M_{01},L_1,\Lambda_\tau)$ and
$N_2 = (P_2,T_2,A_2,C_2,E_2,\mathbb{M}_2,\mathbb{Y}_2,M_{02},L_2,\Lambda_\tau)$, a binary relation $R \subseteq \mathbb{M}_1 \times \mathbb{M}_2$ is a *bisimulation* if for all $(M_1,M_1') \in R$ then the following two conditions hold:

a. for each $z^* \in \sigma(\mu\Lambda)$ and $M_2 \in \mathbb{M}$ such that $M_1 \xrightarrow{z^*} M_2$ there is $M_2' \in \mathbb{M}'$ such that $M_1' \xrightarrow{z^*} M_2'$ and $(M_2,M_2') \in R$

b. for each $z^* \in \sigma(\mu\Lambda)$ and $M_2' \in \mathbb{M}'$ such that $M_1' \xrightarrow{z^*} M_2'$ there is $M_2 \in \mathbb{M}$ such that $M_1 \xrightarrow{z^*} M_2$ and $(M_2,M_2') \in R$

Two labelled CPNs are *bisimilar* if there is a bisimulation relating their initial markings (i.e. if $(M_{01},M_{02}) \in R$).

We now consider the relationship between system morphisms and bisimulation. We have already commented that system morphisms will restrict the possible system behaviours, that is, a refined behaviour will have a corresponding abstract behaviour, but not necessarily vice versa. This requirement is in fact weaker than bisimulation, since it does not require that every abstract behaviour has a corresponding refined behaviour. Therefore an incrementally changed component is not necessarily strongly substitutable (see Section 3.2). If we strengthen the system morphism definition by also requiring that the refined net is *at least as live* as the abstract net (see Definition 4.20) then there is a labelling of the refined net such that a weak bisimulation holds between the abstract and refined net. This is proved in Proposition 4.21.

Definition 4.20. Consider an abstract net N', refined net N, related by system morphism $\phi : N \to N'$. Given $Y_0{}^* \in \sigma\mathbb{Y}$ such that $M_0[Y_0{}^*\rangle M$, and given $(t,c) \in FE$ then we say N is *at least as live* as N' provided:

a. $\forall Y \in \mathbb{Y} : \phi(E^-(Y)) = \phi(E^-)(\phi(Y))$

b. $Y_0{}^*$ can be extended to a complete firing sequence $Y_0{}^*Y_1{}^*$ such that:

1. $M[Y_1{}^*\rangle M_1$ (what we mean by completion)
2. $\phi(Y_1{}^*) = \emptyset$ (minimal completion)
3. $\phi(M_1) = M_1'$ is a reachable marking of N' (by system morphism).

c. if $\phi(M) = M'$ and there is an enabled step Y' of N' from M', i.e. $M'[Y'\rangle$, then there is a complete step sequence $Y^* = Y_1 \ldots Y_m$ of N such that:

1. $M[Y^* >$
2. $\phi(Y_1) = \ldots \phi(Y_{n-1}) = \emptyset$
3. $\phi(Y_n) = Y'$
4. $\phi(Y_{n+1}) = \ldots \phi(Y_m) = \emptyset$

d. if $\phi(M) = M'$ and there is an enabled step Y' of N' from M' leading to M_1', i.e. $M'[Y'\rangle M_1'$ then there is a marking M_1 of N and a complete step sequence $Y^* = Y_1 \ldots Y_m$ of N such that:

1. $M[Y_1 Y_2 \ldots Y_m > M_1$
2. $\phi(M_1) = M_1'$
3. $\phi(Y_1) = \ldots \phi(Y_{n-1}) = \emptyset$
4. $\phi(Y_n) = Y'$
5. $\phi(Y_{n+1}) = \ldots \phi(Y_m) = \emptyset$

Note:

a. The restriction of the tokens removed by a step is the same as the tokens removed by the restriction of the step. This implies that ignored transitions cannot remove tokens from retained places.

b. Each incomplete firing sequence can be extended to a complete firing sequence.

c,d. Each abstract step corresponds to some activity $(Y_1 \ldots Y_{n-1})$ which does not have a correspondence in the abstract net (e.g. some activity internal to supernodes or extensions), followed by some activity (Y_n) which does have a correspondence in the abstract net, followed by further activity $(Y_{n+1} \ldots Y_m)$ which does not have a correspondence in the abstract net (e.g. some activity internal to supernodes or extensions).

In Proposition 4.21 we prove that if a refined net is at least as live as the abstract net then the refined and abstract nets are bisimilar. The proof is joint work between the author and Lakos [117].

Proposition 4.21. If the net N is a refinement of N' by the system morphism $\phi : N \to N'$, and N is at least as live as N', then N is bisimilar to N'.

Proof.
First we define the labelling functions. Let the labelling function, L' of the abstract net N' be such that each distinct firing element has a distinct label. (The identity function would suffice.) We define the labelling function, L, of the refined net, N, such that each firing element that maps to an abstract firing element under ϕ has the label $L'(\phi((t,c)))$, and each firing element for which there is no mapping defined has the label τ.
Thus $L(Y^*) = L'(\phi(Y^*))$.

We now prove N is bisimilar to N' by contradiction. In this proof, we relate markings as follows: if $M_0[Y_0^*\rangle M_1$, and thus $M_0'[\phi(Y_0^*)\rangle M_1'$, then $(M_1, M_1') \in R$. Suppose $(M_0, M_0') \notin R$ where R is as defined in Definition 4.19. Then there are two cases:

a. There exists $z_1^*, z_2^* \in \sigma(\mu\Lambda)$ such that $M_0 \stackrel{z_1^*}{\Longrightarrow} M_1 \stackrel{z_2^*}{\Longrightarrow}$ and $M_0' \stackrel{z_1^*}{\Longrightarrow} M_1'$ but not $M_1' \stackrel{z_2^*}{\Longrightarrow}$. (Note that z_1^* may be empty.)

In this case, since $M_0 \stackrel{z_1^*}{\Longrightarrow} M_1 \stackrel{z_2^*}{\Longrightarrow}$ then there exists $Y_1^*, Y_2^* \in \sigma\mathbb{Y}$ such that $M_0[Y_1^*\rangle M_1[Y_2^*\rangle$.

The system morphism property (Definition 4.7 (c)) implies that $\phi(M_0)[\phi(Y_1^*)\rangle M_x'$ for some $M_x' \in \mathbb{M}'$. We first show that $M_x' \stackrel{z_2^*}{\Longrightarrow}$.

We know from Definition 4.20 (b) that Y_1^* can be extended to a complete sequence $Y_1^* Y_c^*$ where $Y_c^* \in \sigma\mathbb{Y}$ and $\phi(Y_c^*) = \emptyset$ such that $M_0[Y_1^* Y_c^*\rangle M_c$ for some $M_c \in \mathbb{M}$ and hence $\phi(M_c) = M_x'$. (Note that Y_c^* will be empty if Y_1^* is complete.) Now suppose $\phi(M_1) > M_x'$. This implies to reach a marking M_c such that $\phi(M_c) = M_x'$, then $\phi(E^-(Y_c^*)) \neq \emptyset$. However, Definition 4.20 (a) implies $\phi(E^-(Y_c^*)) = \phi(E^-)(\phi(Y_c^*)) = \phi(E^-)(\emptyset) = \emptyset$. Therefore $\phi(M_1) \leq M_x'$. Now we know that $M_1[Y_2^*\rangle$. The system morphism property (Definition 4.7 (c)) implies that $\phi(M_1)[\phi(Y_2^*)\rangle$. Therefore $M_x'[\phi(Y_2^*)\rangle$. We also know that $L(Y_2^*) = z_2^*$. Again the definition of the labelling function then implies $L'(\phi(Y_2^*)) = z_2^*$. Thus we have $M_x' \stackrel{z_2^*}{\Longrightarrow}$.

Now we show that $M_x' = M_1'$. We first observe that since the labelling function L' assigns distinct labels to all firing elements then for any $M_a', M_b', M_c' \in \mathbb{M}'$ and $z^* \in \sigma(\mu\Lambda)$ if $M_a' \stackrel{z^*}{\Longrightarrow} M_b'$ and $M_a' \stackrel{z^*}{\Longrightarrow} M_c'$ then $M_b' = M_c'$. Now, since $\phi(M_0) = M_0'$ then $M_0'[\phi(Y_1^*)\rangle M_x'$. Further, by the definition of the labelling function, since $L(Y_1^*) = z_1^*$ then $M_0' \stackrel{z_1^*}{\Longrightarrow} M_x'$. But we know $M_0' \stackrel{z_1^*}{\Longrightarrow} M_1'$. Therefore $M_1' = M_x'$.

Thus we have shown $M_1' \stackrel{z_2^*}{\Longrightarrow}$. This is a contradiction.

b. There exists $z_1^*, z_2^* \in \sigma(\mu\Lambda)$ such that $M_0' \stackrel{z_1^*}{\Longrightarrow} M_1' \stackrel{z_2^*}{\Longrightarrow}$ and $M_0 \stackrel{z_1^*}{\Longrightarrow} M_1$ but not $M_1 \stackrel{z_2^*}{\Longrightarrow}$. (Note that z_1^* may be empty.)

In this case, since $M_0 \stackrel{z_1^*}{\Longrightarrow} M_1$ then there exists $Y_1^* \in \sigma\mathbb{Y}$ such that $M_0[Y_1^*\rangle M_1$. Further, since $M_0' \stackrel{z_1^*}{\Longrightarrow} M_1'$ then $M_0'[\phi(Y_1^*)\rangle M_1'$, and since $M_1' \stackrel{z_2^*}{\Longrightarrow} M_2'$ then there exists $Y_2^{*\prime} \in \sigma\mathbb{Y}'$ such that $M_1'[Y_2^{*\prime}\rangle M_2'$.

Now by Definition 4.20 (b) Y_1^* can be extended to a complete sequence $Y_1^* Y_c^*$ where $Y_c^* \in \sigma\mathbb{Y}$ and $\phi(Y_c^*) = \emptyset$. (Note that Y_c^* will be empty if Y_1^* is complete.) Now since $\phi(Y_c^*) = \emptyset$ then by the definition of the labelling function, $L(Y_1^* Y_c^*) = L(Y_1^*) = z_1^*$. Suppose that $M_0[Y_1^* Y_c^*\rangle M_c$ where $M_c \in \mathbb{M}$. By Definition 4.7 (d) $\phi(M_c) = M_1'$. Definition 4.20 (c,d) implies that there is a sequence $Y_2^* \in \sigma\mathbb{Y}$ such that $M_c[Y_2^*\rangle$ and

$\phi(Y_2{}^*) = Y_2{}^{*\prime}$. Thus by the definition of the labelling function, $M_0 \stackrel{z_1^*}{\Longrightarrow} M_c \stackrel{z_2^*}{\Longrightarrow}$. Since M_c is reachable from M_1 by actions labelled by τ (i.e. since M_c is reachable from M_1 by $Y_c{}^*$) then $M_0 \stackrel{z_1^*}{\Longrightarrow} M_1 \stackrel{z_2^*}{\Longrightarrow}$. This is a contradition.

We have observed a contradiction in all cases. Therefore $(M_0, M_0') \in R$. That is, N is bisimilar to N'. ◊

We have shown that if N is related to N' by a system morphism, and additionally N is *at least as live as* N' then N is (weakly) bisimilar to N'. The forms of node refinement given in Section 4.6 will not necessarily satisfy the *at least as live as* definition (Definition 4.20). This highlights that the system morphism requirements are weaker than those of bisimulation. As we discuss in the next chapter this means that Incremental CPN Modelling is widely applicable in practice, whereas proposals requiring bisimilarity often cannot be used. We now discuss the implications of the *at least as live as* definition on place and transition refinement as presented in Section 4.6.

The *at least as live as* definition (Definition 4.20) implies that the subnet refinement of the canonical basis of a superplace will not prevent a token that is input to the canonical basis from being offered in the output of the basis. For example, a subnet refinement of the canonical basis of a superplace must not deadlock and thus prevent a token that has been input to the basis from being offered by the basis.

For transition refinement (supertransitions) the restrictions are more severe. This is not surprising since bisimulation is based on observing actions. The general form of transition refinement given in Section 4.6 will not satisfy Definition 4.20 (a). The reason for this is that an input border transition of a supertransition can consume a token from a place that has a corresponding abstract place, but the input border transition is not mapped by the system morphism. For the canonical basis of a supertransition to satisfy Definition 4.20 (a) there must be only one input border transition. Rather than the *switch* transition it is the occurrence of the input border transition that is mapped by the system morphism. (An alternative to mapping the input border transition is to collapse the input border transition and the *switch* transition using a transformation such as those of Haddad [86] (See Section 6.3.8).) Further to this (and similar to place refinement), Definition 4.20 implies that the subnet refinement of the canonical basis of a supertransition must not prevent a received token from being offered. For example, a subnet refinement of the canonical basis of a supertransition must not deadlock and thus prevent a token that has been received from being offered.

4.8 Summary

In this chapter we presented Incremental CPN Modelling. This is a new approach towards incremental change proposed by Lakos [116]. It consists of a general constraint on incremental change — that each refined behaviour must have a corresponding abstract behaviour — which is formulated in terms of morphisms on CPNs. In order to guide the developer to the forms of change that can be used to ensure this principle is followed, three specific forms of refinement were identified: *type refinement*, *subnet refinement*, and *node refinement*. It is expected that these will commonly be used in combination in practical applications.

In the next chapter we assess the practical applicability of type, node, and subnet refinement. We examine several case studies which suggest that, opposed to existing approaches, Incremental CPN Modelling is widely applicable in practice.

Existing proposals generally focus on the substitutability of the incrementally changed component (see Chapter 3). Here the system must not be able to detect any difference between the original and incrementally changed component; this can be ensured using *bisimulation*. In the final part of this chapter we examined the relationship between Incremental CPN Modelling and bisimulation, showing that if additionally the refined net is *at least as live* as the abstract net, then the refined net is bisimilar to the abstract net.

Chapter 5

Incremental CPN Modelling In Practice

Practice is the best of all instructors
PUBLILIUS SYRUS

As we saw in Chapter 3 commonly existing proposals for constraining incremental change focus on the substitutability of the incrementally changed component. Our primary concern with such proposals is that they are not widely applicable in practice. In this chapter we examine the applicability of Lakos's [116] Incremental CPN Modelling in practice.

One concern identified with existing proposals for constraining incremental change was that, in general, they do not guide the developer to the forms of incremental change that are appropriate (see Chapter 3). Instead they simply provide condition(s) that must be satisfied. In contrast to this, Incremental CPN Modelling consists of not only a general constraint to ensure conceptual specialisation, but also three forms of refinement that comply with the general principle (see Chapter 4), and therefore guide the developer to incremental change that is considered appropriate.

Another concern with existing proposals identified in Chapter 3 was that it is often difficult to prove the relation required between the abstract and the refined model holds. Commonly such proposals cannot be statically checked. We argued in Chapter 3 that this was one reason why such proposals have not been adopted in formal languages.

We begin this chapter with a discussion of how Incremental CPN Modelling can be statically checked. Then, in order to assess the appropriateness of the proposed refinements, we examine case studies from the literature. Typically, these studies present the application of a formal method to a real (or realistic) problem. They are therefore constrained by that formal method and its provisions for incremental change. Clearly it is unlikely that pre-existing case studies will use exactly the canonical refinements proposed in Chapter 4.

Nevertheless, it is valuable to compare the refinements used in the case studies with those supported in Incremental CPN Modelling and hence assess the practical applicability of Incremental CPN Modelling. These studies indicate that the Incremental CPN Modelling refinements are widely applicable in practice, whereas strong substitutability often does not hold.

Finally, since the UML has become the de facto industry standard for modelling the dynamic behaviour of an object, we conclude this chapter with a brief discussion of the application of the refinements supported in Incremental CPN Modelling to the Statecharts of the UML. This last section has been previously published [125].

5.1 Checking Incremental Change

In Incremental CPN Modelling, to check whether a refinement is valid involves checking that the definitions 4.9 – 4.14 hold. The important thing to observe is that these definitions are statically checkable. To check type and subnet refinement (definitions 4.9 and 4.11) we must check that the structure of the refined net is valid, that the refined colours are valid, and that the abstract effect of firing a transition with a given mode in the refined net is the same as the effect of firing the corresponding abstract transition (if it exists) with the corresponding abstract mode. Such checks could be performed by a tool as the user makes the refinement, or alternatively, could be performed once all refinements have been made.

We have implemented such checks in the Petri Net tool Maria [136]. We observe that the time required to perform these checks is $O(n)$ where n is the total number of firing modes and token elements in the refined net. We therefore recommend that the number of firing modes and token elements be kept as small as possible. In a tool (such as Maria) which does not allow explicit definition of the firing modes of a transition, this can be achieved by keeping the colour sets of places as small as possible.

To check that node refinements are valid (definitions 4.13 and 4.14), we must first check that the supernode consists of a canonical basis, and second that the refinement of the basis is a valid type and/or subnet refinement. The first check could be easily performed by the tool (or alternatively the tool could only allow node refinement with a canonical basis, meaning this check would not be required). The second check can be made as described above.

5.2 Case Studies

We now examine the incremental change used in several case studies. These studies are examined to assess the applicability of Incremental CPN Modelling constraints. Clearly there will be examples where the refinement used does not conform to Incremental CPN Modelling. However the majority of the studies we have examined are formulated (or can be easily reformulated) to use the Incremental CPN Modelling refinements. This suggests that, opposed to existing proposals for constraining incremental change, the forms of refinement proposed in Incremental CPN Modelling are widely applicable in practice.

The studies we examined were chosen from those relatively few studies available that provided enough information to make a detailed study of the incremental change

used. These studies are: the HTTD production cell study [41], the Graftab specification of a Steam boiler Controller [179], the Cooperative Structured Editors model [29], the Hydro-Electric Corporation's Billing System [37], an Artificial Neuron Network lifecycle [62], the implementation of a dataflow language [112], the Sliding-Window Protocol [113], the specification of services in an intelligent network [40], a Communications Gateway [74, 73], the Z39.50 Protocol [127, 123, 122], the Fieldbus Protocol [143], a Die Bonder Study [101], and an Air-to-Air Missile Simulator [82, 83]. Clearly it is not practical to present a detailed examination of all of the studies. The studies we present here have been chosen since they: form a representative sample of those studies examined, are industrial studies, and are published in international conferences or journals. In Section 5.2.6 we provide a table summarising the types of changes made in all the case studies examined. In our presentation we do not provide declarations or functions for the nets since the refinement made can be conveyed to the reader without these. We refer the reader to the cited publications for full details of the nets, including declarations and functions.

Some of the case studies we examined were formulated using the Design/CPN tool [105] which supports substitution transitions as a means of abstraction but without behavioural constraints. Substitution transitions are like macros or textual substitution: they maintain structural compatibility, but there is no concept of abstract behaviour. The semantics of the construct are defined in terms of textual (or graphical) substitution. This means that even if the designer has in mind a notion of abstract firing, there is no way to capture this formally in the model, either as an aid to understanding or as a hint to improve analysis.

5.2.1 Designing and Verifying a Communications Gateway Using CPNs

The concept of a packet radio network has been developed to allow military users deployed in the field to get access to civilian services and to integrate military communications at all levels of command and control with the civilian infrastructure. However, civilian telecommunication networks are increasingly broadband networks that offer a large number of services. Floreani et al [74, 73] have used the Design/CPN tool [105] to specify and verify a gateway between a narrowband packet radio network and a broadband-ISDN network. The design of this gateway is complex, partly due to its distributed nature, and therefore, although the CPN formalism does not directly support incremental specification, the methodology employed in the case study involved repeated use of incremental specification.

The most abstract model is given in Figure 5.1. This model describes the basic mechanics of the model. It consists of the sending and receiving interfaces of the gateway, a service handler that manages specific end user service (voice, data, messaging) transfer through the gateway, and the Call Control Application (*CCA*). The *CCA* is the entity where call control is mapped between the two networks. In the most abstract model the *CCA* was represented simply as a place. This meant that "... fundamental problems such as call setup and release issues could be resolved at higher levels before the intricacies of the lower levels clouded the issues." [73, p. 73]. This is exactly the situation we see as typical of incremental specification.

The transitions *send_prims*, *rec_prims*, and *sh_prims*, are substitution transitions that handle the setup and release of calls, and configuration of the service handler, respectively. As noted previously, the substitution transitions supported by Design/CPN are like macros: they maintain structural compatibility, but not behavioural compatibility. However, in

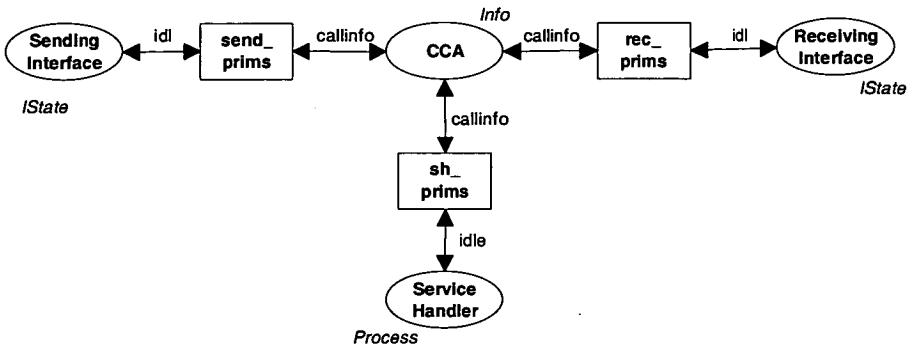


Figure 5.1: Abstract model of a communications gateway

this case study, these particular substitution transitions do satisfy the requirements for transition refinement, and hence the explicit use of transition refinement would increase the clarity of the model and guarantee behavioural compatibility between refinement and abstraction.

Additionally, in this abstract model Floreani defines a colour set containing the information required for setting up a call, and a colour set with this information plus addressing information. It is clear that this latter colour set is a type refinement of the former.

Once the interfaces were verified to behave correctly, the abstract model was further refined. The *CCA* was replaced with a subnet that models sending, receiving, and gateway call control. The refined net is shown in Figure 5.2. Once again the transitions of this model are substitution transitions which again satisfy the Incremental CPN Modelling requirements for transition refinement.

It is useful to briefly explain the operation of the *CCA*. Call setup messages, are passed to the Sending Call Control (*SCC*). If the call is accepted for further processing (depending upon interaction with the Sending Resource Management (*SRM*)), the *SCC* requests a gateway connection from the gateway call control by passing a token to *GCCIO*. The gateway call control queries the node location process and the gateway resource management process to look up routing information and evaluate the available gateway resources respectively. If the destination address can be found and there are enough resources available within the gateway, then the gateway informs the service handler. This involves setup, negotiation of mapping parameters, and the user service mapping within the gateway. If successful, the gateway passes the call setup information to the Receiving Call Control (*RCC*) for transfer to the end user. If the end user accepts the call, a connect message is passed back to the calling user and the user service, such as file transfer, can commence.

In Figure 5.3 we have modified the net of Figure 5.2. The place *SCC_RCC_GCCIO* has been added, as have transitions connecting this place to the *SCC*, *RCC*, and *GCCIO* places. The *SCC_RCC_GCCIO* place has a type equal to the union of the types for the *SCC*, *RCC*, and *GCCIO* places. The *SCC_RCC_GCCIO* place serves as a combined input and output place for the gateway. That is, the requests and responses of the gateway are input to or taken from the *SCC_RCC_GCCIO* place rather than being input directly to the *SCC*, *RCC*, or *GCCIO* place as appropriate. By replacing the *SCC_RCC_GCCIO* place of this modified net with *input*, *output*, and *buffer* places and associated *accept* and *offer* transitions we can obtain a canonical node refinement. Although (a refinement of) the additional place *SCC_RCC_GCCIO* is required under the constraints canonical refinements, this place is actually redundant. It is therefore clear that the behaviour of the net

of Figure 5.3 is equivalent to the behaviour of the net of Figure 5.2 and hence the gateway refinement is equivalent to a canonical node refinement.

We observe that in the abstract model, when a connect indication is made, or setup requested, the appropriate token will always be accessible from the gateway place. However, this is not the case in the refined model, where a connect indication can be requested and fail because resources are not available, or if the destination address cannot be found. Similarly, a connect setup request can fail because not enough resources are available. Thus not every behaviour of the abstract net will have a corresponding behaviour in the refined net. That is, the refined net does not satisfy the strong substitutability requirements of the proposals in Chapter 3.

In his Ph.D. thesis, Floreani [73] uses trace equivalence to relate the abstract and refined model, which seems an appropriate consistency relationship between the service and protocol specifications of a protocol. However, we note that the relationship can also be captured by our forms of incremental specification. The relationship between these two approaches is a subject for further research.

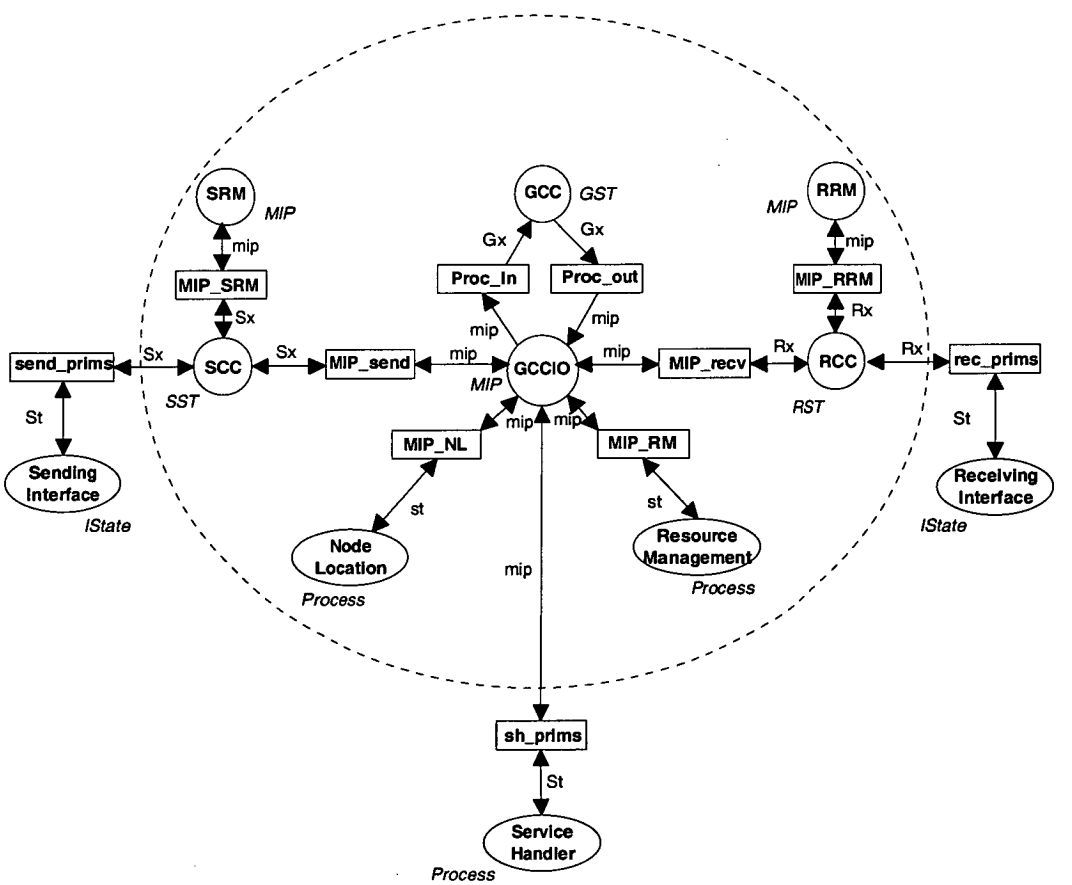


Figure 5.2: Refinement of the communications gateway

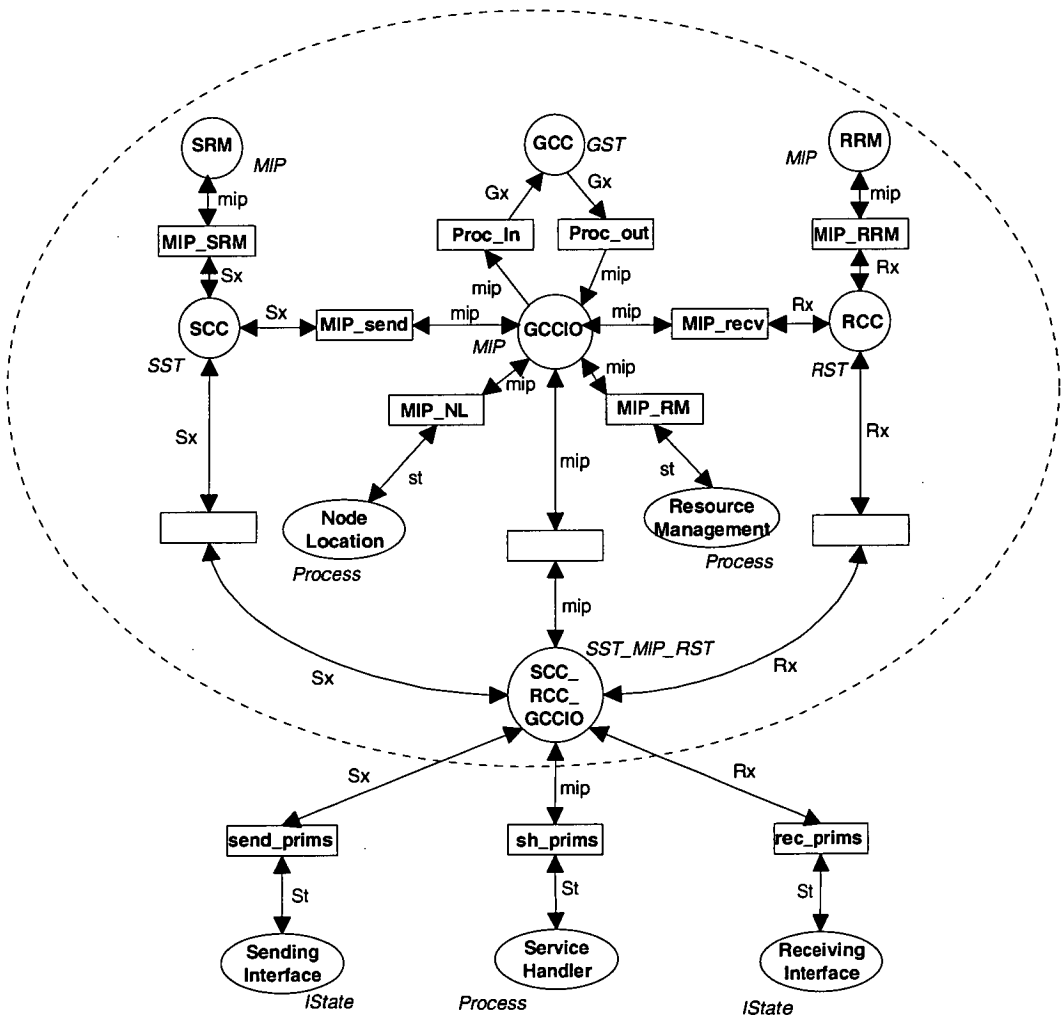


Figure 5.3: Equivalent refinement of the communications gateway

5.2.2 The Z39.50 Protocol

The intent of the Z39.50 Protocol for Information Interchange [13] is to provide the kernel of a client/server system that allows computer-to-computer information search and retrieval. The standard was developed to overcome the problems associated with multiple database searching such as needing to know the unique menus, command language, and search procedures of each system accessed.

Lakos and Lamp [127, 123, 122] have modelled both the 1992 and 1995 versions [12, 13] of the Z39.50 protocol. The Z39.50 standard defines the operation of what is called the Z39.50 origin and target, which are those parts of the client and server respectively that provide the facilities associated with networked information search and retrieval. In order to capture the structure more clearly, Lakos and Lamp model a number of origin-initiated services (including *Initialize*, *Release*, *Search*, *Present*) using a particular style of subnet shown in Figure 5.4. The *state* place indicates the state of the subnet. The subnet can be in one of three states: an initial state, an intermediate state, and a final state. In this net, when there is a request message to transmit and the protocol entity is in its initial state, the transition *send* sends the request from the origin to the net. The *receive* transition can fire if the appropriate response is received from the net and the protocol entity is in its intermediate state. The appropriate indication message is then sent to the protocol user and the protocol entity enters its final state.

Similarly Lakos and Lamp model a number of target initiated responses with a generic net. The Z39.50 protocol can then be specified by using an instance of the request service subnet for each of the different possible requests (including *Initialize*, *Release*, *Search*, *Present*), and an instance of the response net for each of the possible responses.

The information to be stored in each message is defined in the Z39.50 specification. For example, a search request (i.e. a request sent by the origin to query databases at the target) includes the names of the databases to be queried, the actual query, etc. However, this information does not affect the basic operation of the Z39.50 protocol, and therefore Lakos and Lamp have used a minimal message format which does not include such information. It was intended that this information would be included in a more detailed model. This would correspond to the use of type refinement.

Access control introduces an access rights challenge into the middle of the normal request-response interaction. Access control has been incrementally added to the generic nets. The request service enhanced with access control is shown in Figure 5.5. The messages of the original net are extended to include access requests and responses, and the

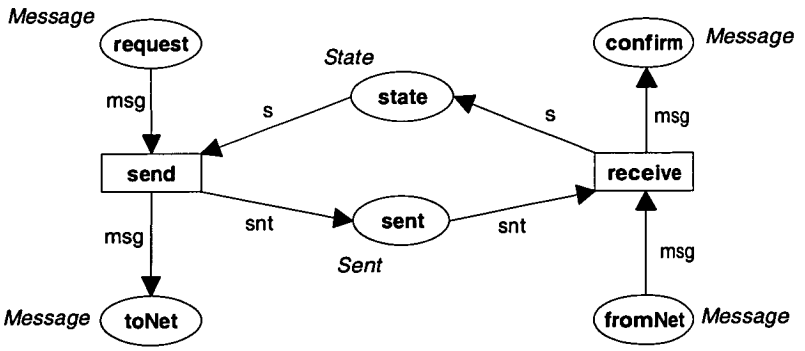


Figure 5.4: Z39.50 request service

subnet is enhanced with the ability to respond to an access control (the added components are rendered with thicker lines).

An access challenge sent from the target will be received in the *fromNet* place. If a request appropriate to such a challenge has previously been sent by the origin, then the *receive challenge* transition will remove the challenge token from the *fromNet* place and add a token to the *access challenge* place (and to the *confirm* place, to indicate a challenge has been received). The *send response* transition then retrieves both the challenge (from the *access challenge*) and the appropriate response (from the *request* place) and sends the response to the target (by placing it in the *toNet* place).

The refinement shown in Figure 5.5 does not qualify as a subnet refinement, since subnet refinement requires that every refined behaviour has a corresponding abstract behaviour. This is not the case since the *receive challenge* transition extracts a sent message from the *sent* place. (Recall that under subnet refinement, Definition 4.11, a transition added by subnet refinement can refer to, but cannot permanently modify, an original place.)

However, access control refinement can be formulated using subnet refinement by introducing a duplicate *Sent* place as shown in Figure 5.6. Since the duplicate sent place is introduced by subnet refinement then the stringent requirements for existing places no longer apply.

Since an access control challenge may fail then not every behaviour of the abstract net will have a corresponding behaviour in the refined net. That is, the incremental addition of access control will not satisfy the strong substitutability requirements of the proposals in Chapter 3.

The protocol entity was further refined in order to support the *abort* service, which allows for abrupt termination. The abort service is not acknowledged and can be initiated by either the target or origin. Once again it is possible to model this refinement using subnet refinement, but not the stronger proposals of Chapter 3.

In addition to these incremental changes, Lakos and Lamp consider whether the later version of the protocol (the 1995 version) can be expressed as a modified version of the earlier one (the 1992 version) [122]. One improvement of Z39.50-1995 over the Z39.50-1992 is the support for segmentation of retrieval responses. Once a search has taken place,

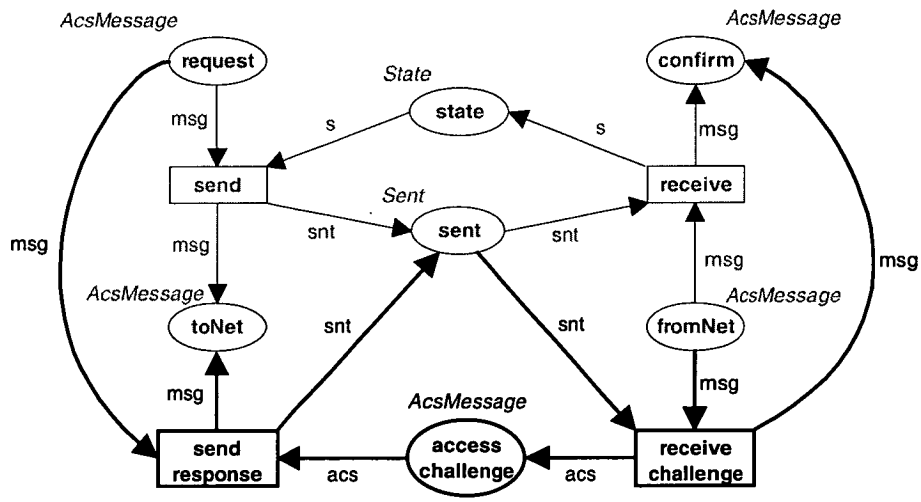


Figure 5.5: Z39.50 request service with access control

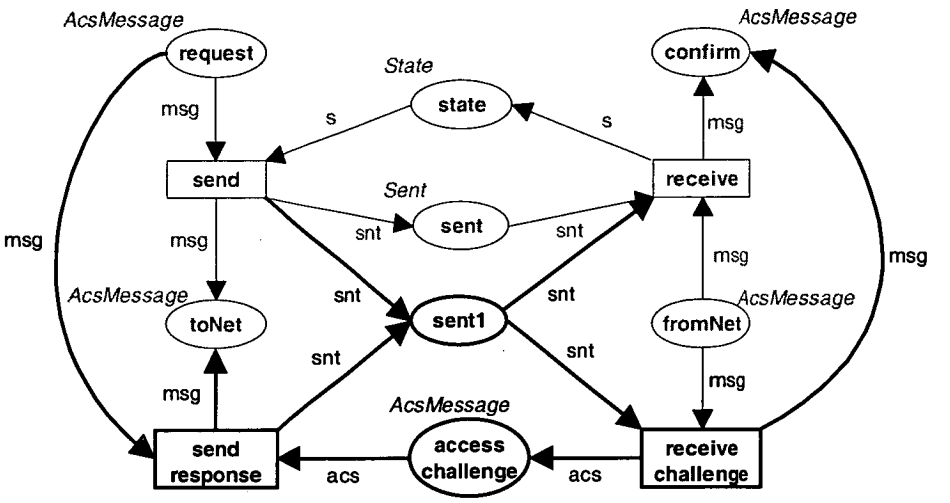


Figure 5.6: Z39.50 request service with access control achieved using subnet refinement

the result set is established and the actual records can then be retrieved. Given the variability of the length of the records, it is possible to return multiple records in each response (if the records are relatively short), or only part of a record (if the records are relatively long). Provision for the first is referred to as level 1 segmentation, while additional provision for the second is referred to as level 2 segmentation. Both are accomplished by the target sending a number of *Segment* requests followed by the *Present* response. (These are referred to as *Segment* requests rather than responses in order to fit the request-response paradigm of the protocol.)

This extended service has been modelled as subnet refinement of the basic request service, as shown in Figure 5.7 (again the newly added components have been rendered with darker and thicker lines).

Another significant change from the 1992 version was to introduce concurrent processing of requests. In the 1995 version, the services still have the same sequence of operations. The difference is that access to the token of the *state* place should not prohibit other services from accessing this token. To accommodate this the *state* place was replaced with the subnet shown in Figure 5.8. Now the place *avail* contains a single token that records a list of the available services. This place is updated each time a request is sent or received. The tokens in the place *active* are copies of the services that have been offered to the environment. This refinement does not qualify as a place refinement since tokens are not conserved. That is, when a token is taken by the environment, (by firing the *get* transition), a new token is also deposited in the *avail* place.

The fact that this change does not qualify as a place refinement is not too surprising given the radical nature of the change. This change demonstrates that the use of inheritance can lead to new components which are not behaviourally compatible with their original component.

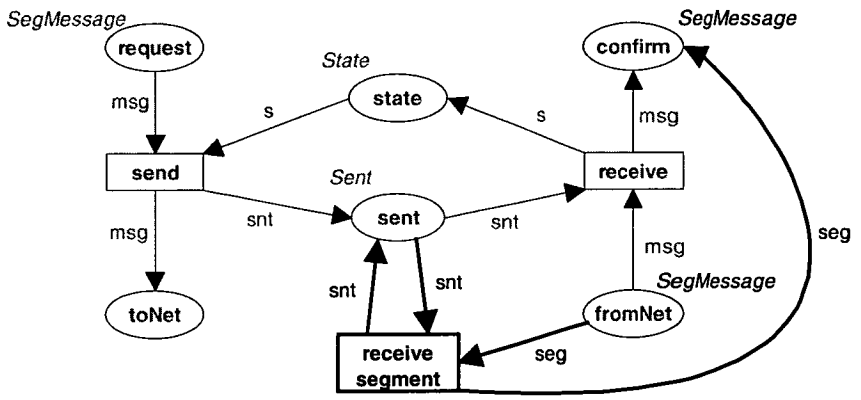


Figure 5.7: Z39.50 request service with segmentation

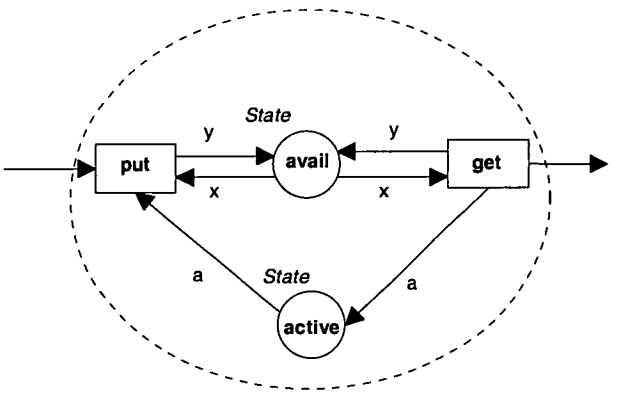


Figure 5.8: The subnet to support concurrent operations

5.2.3 Fieldbus Protocol

To meet new challenges in factory automation and process control, new local area network architectures, called Fieldbus Networks, have been proposed. The proposed architectures are hierarchical, where floor level devices (sensors and actuators), at low speed, are linked to their controlling devices and controlling devices can be linked among themselves at higher levels and higher speeds. The International Society for Measurement and Control (ISA) Data Link Layer (DLL) protocol has been proposed to handle such distributed control, and has been modelled using Design/CPN [143]. The top-level view of the system is shown in Figure 5.9.

All the transitions in that view are substitution transitions and capture the logic of the various devices. All the places are simple places which are used to convey delegation of commands and associated time periods to the devices. The *LAS-Model* captures the logic of the *Link Access Scheduler* (LAS) with regard to arbitrating the synchronous and asynchronous delegation of commands and time slices to the various devices. The *LAS-DLE* is the *Data Link Element* (DLE) for the *LAS*, which therefore captures its own requirements for time slices. Three instances of sensors (which return information about device status) are indicated, as are three instances of actuators (which receive data to modify the device status).

Time slices are allocated regularly to the devices according to their fixed requirements. This is known as the synchronous phase. Time is also allocated as available and as required for miscellaneous tasks such as data collection and error reporting. When the *LAS-Model* wishes to perform the synchronous token delegation for a sensor, it deposits a token in the place *CD1* (for *Compel Data 1*) and then waits until the sensor relinquishes control and returns the relevant status information by depositing a token in the place *DT1* (for *Data 1*). The synchronous token delegation for an actuator only requires data to be sent (i.e. no response is expected) and hence is achieved by the *LAS-Model* depositing a token in the place *CD2* (for *Compel Data 2*), whereupon the *LAS-DLE* deposits a token in the place *DT2* (for *Data 2*) that is consumed by the relevant actuator. The *LAS-DLE* also deposits a token in the place *End-DT* to indicate the time when the actuator will be ready.

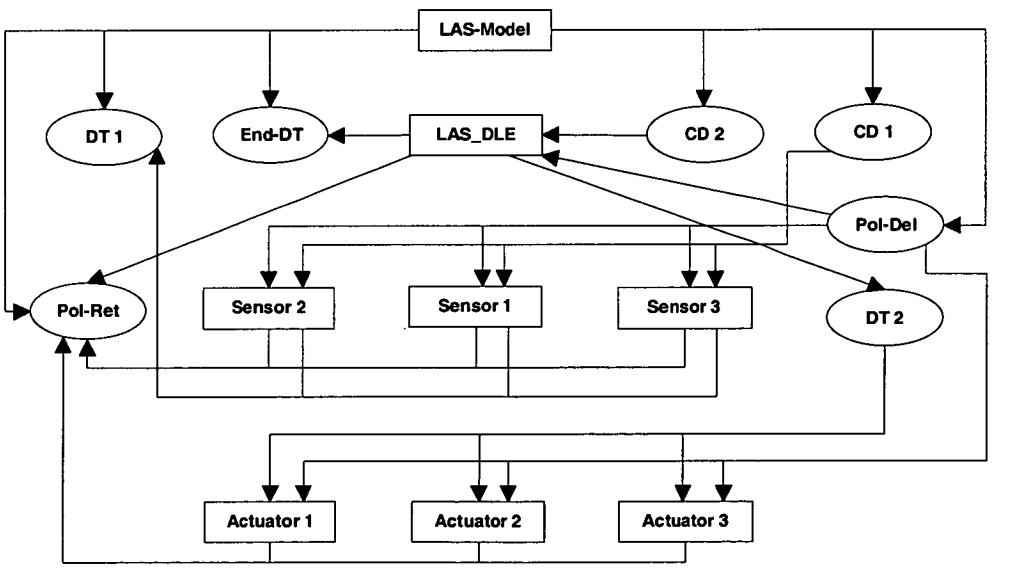


Figure 5.9: Abstract view of the fieldbus protocol

When the *LAS-Model* wishes to perform the asynchronous token delegation for sensors, actuators, or even itself, it deposits a token into the place *Pol-Del* and waits till a token is returned into the place *Pol-Ret*. Each device has a rather complex subnet for computing the time consumed out of the asynchronous time delegation and the possible further requests for time.

The key point is to recognise that the above abstraction captures a lot of information about the behaviour of the system. Sensors and actuators have the notion of abstract firing in two different sets of modes. One set of modes relates to the synchronous token delegation, and the other set of modes relates to the asynchronous token delegation. Each one of these consumes a token and produces a token after some internal activity. (Actuators are slightly different in only consuming a token during the synchronous token delegation.) Thus, the substitution transitions for sensors and actuators basically satisfy the criteria of Incremental CPN Modelling for transition refinement. The abstract view will probably allow an arbitrary order of token delegation, while the refinement will constrain that order by the internal logic of the various components.

However, as presented in the original paper, these substitution transitions exhibit further complexities. Firstly, they share information via place fusion. This is not supported by Incremental CPN Modelling, but the same effect could be achieved by passing this information via additional interface places. Secondly, the system incorporates the notion of simulated time so that simulations can collect performance results. It would be useful to incorporate time into Incremental CPN Modelling.

5.2.4 Modelling a Die Bonder with Object-Oriented Timed Petri Nets

Janneck [101] uses an OO Timed PN language [71] to model a die bonder. A die bonder is a manufacturing machine used in the integrated circuit packaging process to glue silicon dies onto lead frames.

The model presents a number of incremental changes, including the modelling of the *PickAndPlace* component, which is one of three main components in the die bonder model. The *PickAndPlace* component synchronises the picking of a new die (from the wafer table) and the bonding of the die onto the lead frame. An abstract model of the *PickAndPlace* component is shown in Figure 5.10[1]. The component receives a ready signal from the wafer table (in place *WHready*). After the die is positioned it is indicated that the die has been picked (*DiePicked*). When the lead frame is in position the component receives a ready signal (in *IndexerReady*) and after the bonding time has elapsed, the component indicates that the die has been bonded (by placing a token in *DieBonded*).

The abstract model is then refined as shown in Figure 5.11. Here each of the picking and bonding transitions are refined into two transitions, shown with a dashed outline. Such a refinement could be achieved using the notion of transition-refinement. However, the motivation behind this refinement was to split the delay time for picking and bonding into two parts, one being fixed by the necessary mechanical movements, and one being a process parameter that can be configured on the machine. The inclusion of time in Incremental CPN Modelling is a matter for further work.

[1] Note that the language used [71] includes a notion of *interfaces*, but to avoid introducing further notation we simply use places to represent these interfaces. Also note that some places have not been named since they are not named in [101].

The final refinement of the *PickAndPlace* component is shown in Figure 5.12. The arc with a circle at one end is an *inhibitor arc* which tests for the absence of tokens. This refinement takes into account that the moving arm sometimes has to be stopped whenever the image acquisition for the optical positioning of the wafer table is not yet complete. This happens when the arm is faster than the ready signal from the wafer table. As it stands, this refinement is not valid according to Incremental CPN Modelling. However, essentially this refinement introduces new behaviour for when the die is ready to be picked, and a node refinement of the *ReadyToPick* place can achieve the same effect. (Incremental CPN Modelling does not consider inhibitor arcs, but any coloured net with inhibitor arcs can be transformed to a behaviourally equivalent coloured net without them [49]).

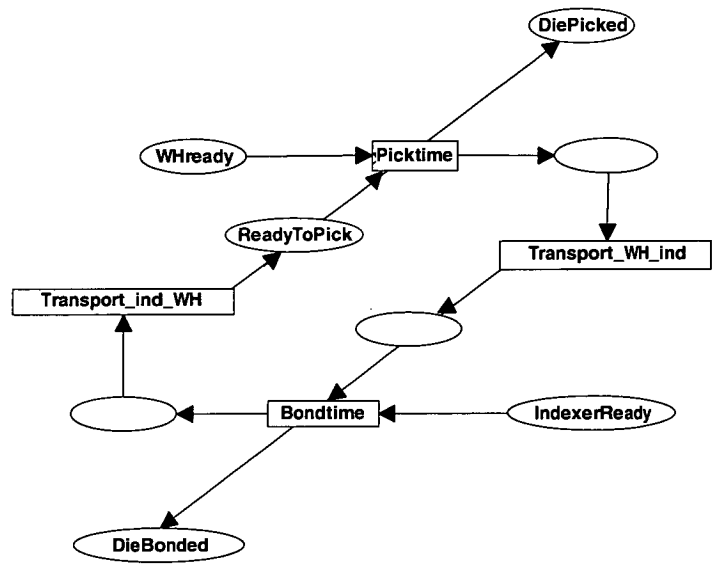


Figure 5.10: An abstract model of a Die Bonder

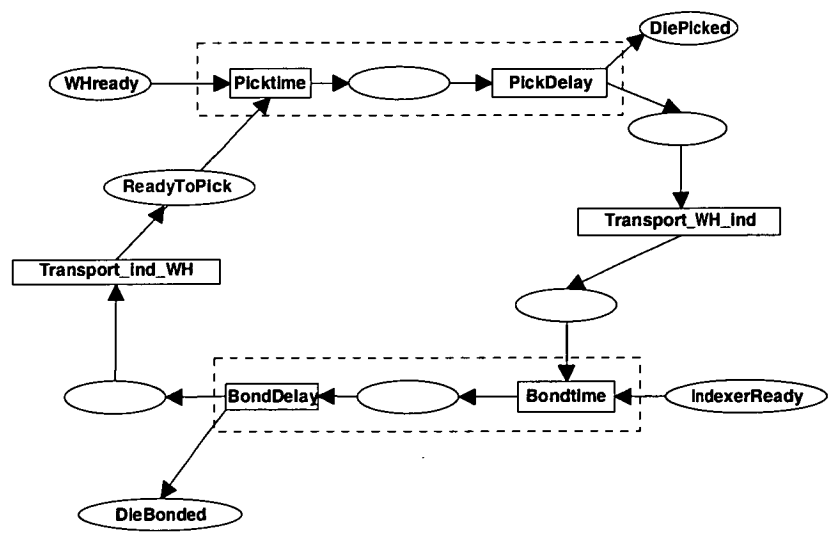


Figure 5.11: The first refinement of the Die Bonder

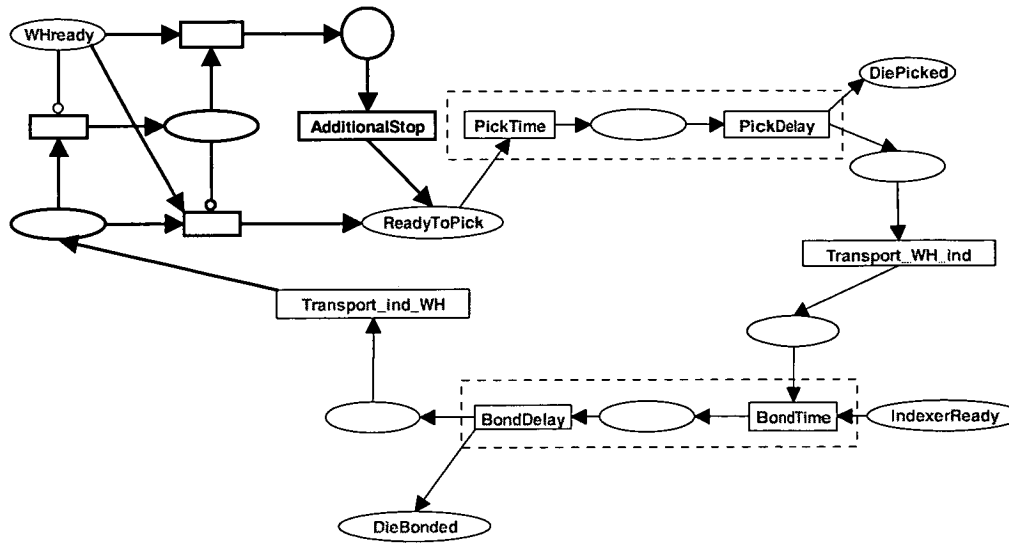


Figure 5.12: The second refinement of the Die Bonder

5.2.5 Air-to-Air Missile Simulator

Gordon and Billington [82, 83] have used Coloured Petri Nets and the Design/CPN tool [105] to design a distributed air-to-air missile simulator, intended to be used as a platform for testing missile guidance and control algorithms. The physical system modelled consists of an engaging aircraft, with an air-to-air missile on board, and a target aircraft. When the engaging aircraft detects the target, it launches the missile. After launching, the missile uses its own guidance system to track the target. The model simulates from when the missile is launched. It does not simulate the launching aircraft or launching procedure.

The missile has two physical devices that detect the location of the target: a radar (RF), and an infrared sensor (IR). Both mechanisms are used to improve accuracy. For example, the radar can give inaccurate data if electronic counter measures are taken by the target. In this case infrared data will be used.

The methodology used in this case study involved first developing an abstract model and then refining this model. The abstract model is shown in Figure 5.13. It has two parts: the graphical user interface (GUI), and the simulator. In this abstract model, the simulator is represented simply by the *Simulate* transition.

The abstract model is refined to include the details of the simulation algorithm. The refined model is presented in Figure 5.14. (Note that to avoid clutter we have not described the colours of the nodes of the refined model nor included a description of the functions. Gordon [82] describes the colours and functions.) The refinement consists of node refinement of the *Simulate* transition, node refinement of the *Outputs* place, and type refinement to introduce three dimensional coordinates to describe the position and velocity.

The refined *Outputs* place consists of four places — *RFRange*, *IRRange*, *TargetToGUI*, and *MCToGUI*) — each of which serves to transfer tokens from the refined *Simulate* transition to the GUI. To avoid crossing arcs, we have duplicated these places in the refined *Simulate* transition subnet. The refined *Simulate* transition consists of four main components: *Target*, *Radar*, *Infrared*, and *Missile Control*. Each component has been enclosed in a dashed box in Figure 5.14. The purpose of each of these components is to calculate new target, radar, infrared, and missile positions and velocities respectively.

The refined model is started by the occurrence of the *Start* transition and the change to the target made by the user is sent to the refined *Simulate* transition. Firstly the *Target* component calculates the new actual position of the target. This new position is subsequently used in the *Radar* and *Infrared* components where tracking algorithms estimate the position of the target. Using these estimated values, the *Missile Control* component estimates the missile position and missile trajectory. Finally, the missile position, radar and infrared values, and target position are relayed to the GUI, allowing either a *Miss* or *Halt* to occur. The occurrence of *Miss* starts a new simulation, whereas *Halt* will stop the simulation.

The net of Figure 5.14 is slightly different from that presented by Gordon and Billington since their model does not use canonical node refinement. In general the canonical construction of a refined place is required to ensure that the general requirements of Incremental CPN Modelling hold (namely that every refined behaviour has a corresponding abstract behaviour). However, in the case of the refinement of the *Outputs* place, since the abstract place is simply replaced by four places then clearly every refined behaviour has a corresponding abstract behaviour, and the canonical construction is not required. For the refined *Simulate* transition we have added the basis places, transitions, and arcs. That is we have added the received, and send (*recd*, *send1*, *send2*, *send3*, *send4*), and the *begin* and *switch* transitions. In Figure 5.14 thicker lines have been used to highlight these added components. By adding these components the supertransition of Figure 5.14 satisfies the requirements for canonical transition refinement (Definition 4.14). The added components do not constrain the occurrence of any transition internal to the refined *Simulate* transition (except that the output border transitions must occur after the *switch* transition). Therefore every firing sequence in the model we present is equivalent to a firing sequence in the model of Gordon and Billington, where the occurrence of the *begin* and *switch* transitions in the refined sequence are hidden.

We note that in the abstract model of the missile simulator there is a non deterministic

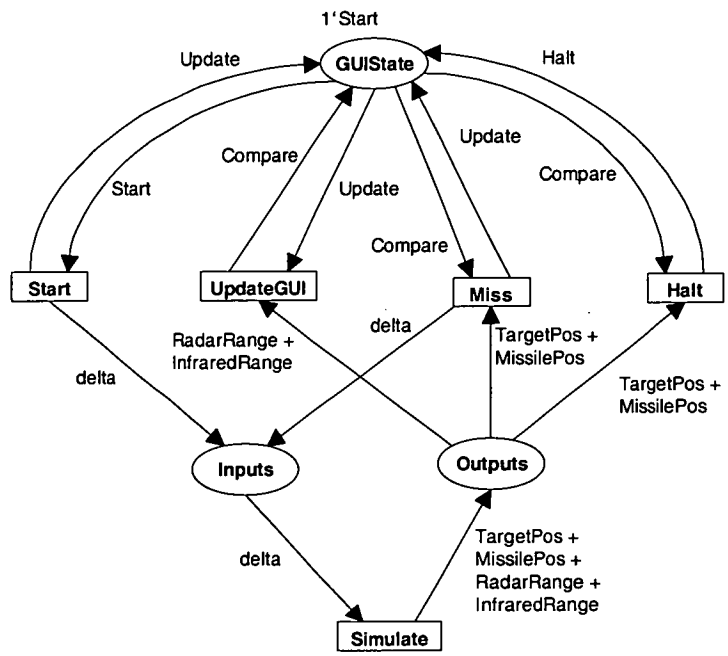


Figure 5.13: Abstract model of a missile simulator

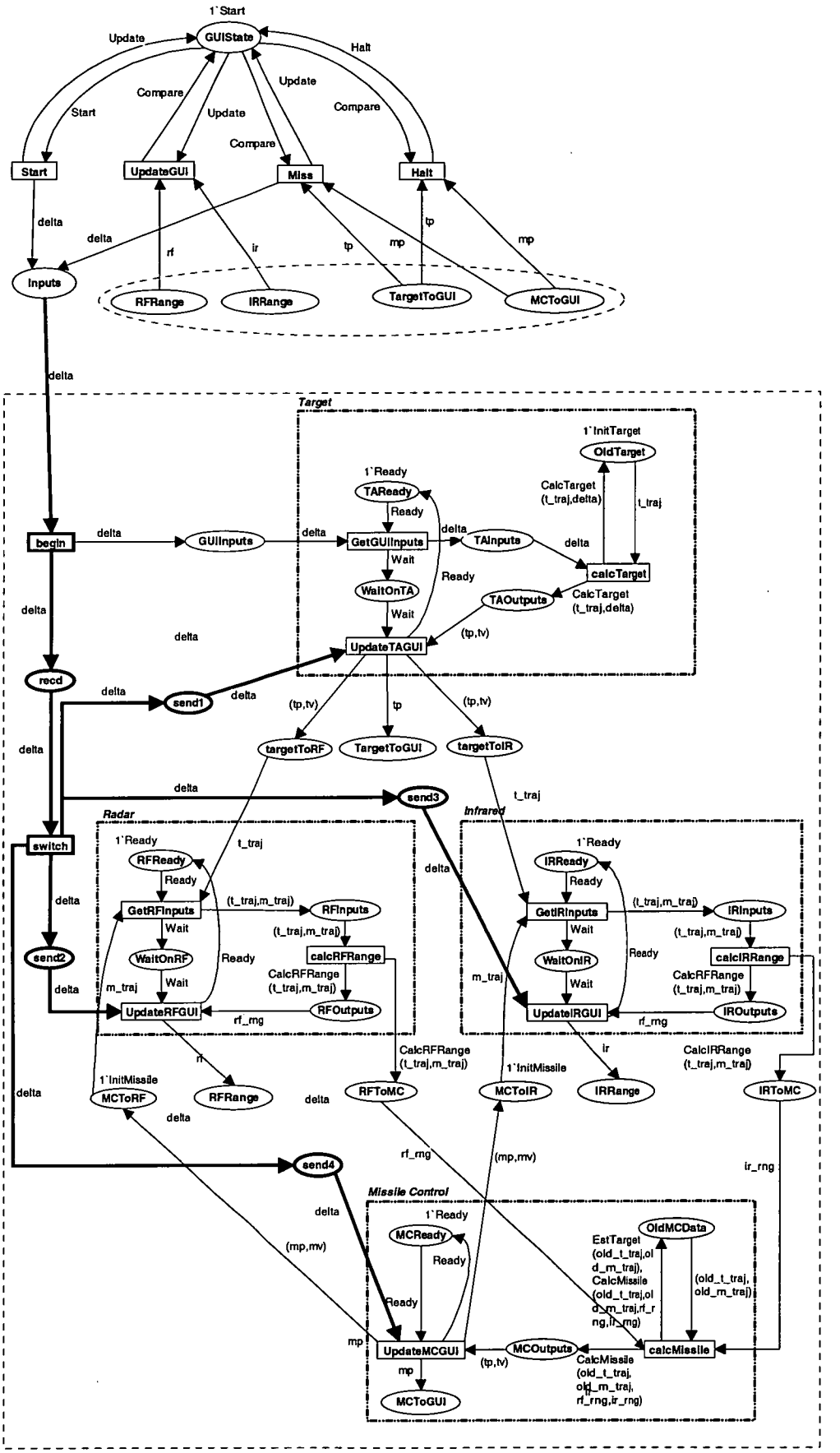


Figure 5.14: Refined model of a missile simulator

choice as to whether the missile will hit the target. However, the introduction of coordinate data means that the choice becomes deterministic and hence the missile may never hit the target. Thus refinement does not satisfy strong substitutability requirements of other proposals.

5.2.6 Case Study Summary

Clearly proposals for incremental change are of little value in practice if they need to be consistently broken. We have examined several case studies for their use of incremental change, and observed that the forms of refinement supported in Incremental CPN Modelling are commonly applicable in practice, whereas the strong substitutability often does not hold. Table 5.1 gives a summary of the refinement used in the various case studies we have examined. The *type*, *subnet* and *node* columns indicate type refinement, subnet refinement, and node refinement were used respectively. The *other* column indicates that the refinement used is not valid under Incremental CPN Modelling. The column *Not strong subst.* indicates that weak bisimulation does not hold between the original and incrementally changed model. Strong substitutability, is required by many existing proposals for constraining incremental change (see Chapter 3), and therefore this column indicates the case studies whose incremental change would not be valid under most existing proposals.

Case Study	Type	Subnet	Node	Other	Not strong subst.
Communications Gateway	✓		✓		✓
Z39.50 Protocol	✓	✓	✓	✓	✓
Fieldbus Protocol			✓	✓	✓
Sliding Window Protocol	✓				✓
Die Bonder		✓	✓		
Cooperative Editors		✓			
Missile Simulator	✓		✓		✓
HTTD production cell			✓		
Steam boiler			✓		✓
HEC Billing System	✓	✓			
Neuron Network lifecycle	✓	✓			
Dataflow language			✓		✓

Table 5.1: Summary of case studies

5.3 Incremental CPN Modelling Applied to the UML

The concern for the practical application of constraints on incremental change has also led us to consider the implications of Incremental CPN Modelling in the UML [125].

As noted in Chapter 2, in OO modelling it is common to describe dynamic behaviour using a Petri Net or State Transition Diagram. Coloured Petri Nets provide a formal and powerful modelling framework. Coloured Petri Nets have a lot in common with the state machines mandated for use in the UML. There is a natural correspondence between state components and between state transition components in the two models. The inclusion of actions in a UML statechart is more involved but can be captured by token modification and additional transitions [114].

There are two significant differences with respect to state components. Firstly, the tokens resident in CPN places capture all the data of a system, whereas the data associated with a UML state machine is captured in three ways: as the current state of the machine, as the globally accessible data values associated with the object, and as the events in transit between state machines. Clearly, CPNs will wrap the first two together into one set of places, and will have a separate set of places to support event transmission. Secondly, multiple tokens may correspond to multiple objects with the same lifecycle, whereas state machines normally assume one associated object. This is not a significant issue since the two views just represent different ways of folding Petri Nets [113].

The correspondence between state transition components is even more direct, given that a similar notation is adopted for so-called concurrent transitions in UML. The bar notation is only used for such transitions in UML, while they are always used in the Petri Net context. While UML transitions do not directly specify firing modes, nevertheless the event parameters and the access to the object state correspond to the CPN transition firing modes. It is also worth noting that the form of state machines supported by UML are more constrained than the possibilities available as CPNs. This is primarily apparent in the restrictions on concurrency (see Appendix A.2).

Given the above correspondences between CPNs and object lifecycles, it is appropriate to use the above forms of refinement to maintain behavioural consistency between the lifecycles of superclasses and subclasses, within the constraints of UML state machines, as detailed below.

Type refinement allows the replacement of types by subtypes provided that the subtype values can be projected onto supertype values. As noted above, such data values will arise as the value of an object and as event parameters. Thus, this form of refinement translates into UML as maintaining consistency between the refined and abstract value of an object, and by maintaining behavioural consistency between a refined and abstract lifecycle in the presence of refined events. Typically this will mean that additional attributes can be added, or that the type of existing attributes can be changed to subtypes, and that the structure of the state machine remains the same.

Subnet refinement allows the addition of states and transitions provided that a refined transition sequence can be mapped to an abstract sequence by ignoring the additional components. In the UML, this means that additional concurrent substates can be introduced provided that the associated transitions respond to (and produce) new events[2]. Similarly, new self loops can be introduced (for new events), even if these involve a sequence of new states and new transitions. The important thing here is that the occurrence of that loop returns to an original abstract state without having traversed other abstract states or processing preexisting abstract events.

Node refinement allows a state to be refined into a hierarchical state with an arbitrary number of components. In UML, this corresponds to replacing a simple state with a complex state. The key issue is that the refined state always has a corresponding abstract state. This is already guaranteed by the semantics of such hierarchical states. That is, the well-formedness rules for complex states (see Appendix A) are sufficient to guarantee the required relation between the substates and the complex state. Similarly, node refinement allows a transition to be refined into a sequence of transitions. (A concurrent transition can

[2]The constraints on concurrency in the UML will mean that the transition must enter one of these newly added concurrent substates when it enters the complex state containing them.

be refined into a more complex set of transitions.) Each complete firing sequence of such a sequence will correspond to a firing of the original abstract transition.

5.4 Summary

As we saw in Chapter 3 it is common for existing proposals that constrain incremental change to focus on the substitutability of the incrementally changed component. Our primary concern with such proposals is that they are difficult to use in practice. Incremental CPN Modelling embodies a general principle to ensure conceptual specialisation, and three forms of refinement that comply with the general principle (see Chapter 4). In this chapter we examined the practical applicability of Incremental CPN Modelling.

One concern we identified with existing proposals for constraining incremental change was that it is difficult to prove the proposed relation holds. Commonly such proposals cannot be statically checked; even automated non-static checks are often infeasible. We argued in Chapter 3 that this was one factor limiting the practical value of such proposals. We began this chapter with a discussion of how the Incremental CPN Modelling refinements can be statically checked. We then examined several case studies that use incremental change. The majority of the incremental change observed can be achieved using Incremental CPN Modelling. In some cases changes were required to the models, but in a proper tool this would not be the case. In contrast, other proposals requiring strong substitutability are often such that the incremental change used in the case studies would not be allowed. Our assessment is therefore that Incremental CPN Modelling is widely applicable. We believe that its use would help to clarify the models and guarantee the conceptual specialisation which is in the mind of the developer, as well as guide the developer to the appropriate forms of incremental change, and (as is shown in the second part of this thesis) provide a mechanism to improve the state space construction and therefore help alleviate the state space explosion problem.

Part II

Incremental Analysis

Chapter 6

State Space Reduction Methods

State explosion cannot be cured without losing some analysis capability
ANTTI VALMARI

A major advantage of formal methods is that they allow for formal reasoning. State space methods (SSMs) are one of the most promising formal reasoning techniques, but they suffer from the state space explosion problem. This chapter introduces SSMs and explores some of the various approaches that have been proposed to alleviate state space explosion.

6.1 Formal Reasoning Techniques

There are two main approaches to verification, analysis, validation and error detection using formal methods: *state space* methods and *theorem proving* methods [194].

Theorem proving methods are based on formulating a correctness claim as a mathematical theorem, and either manually or with the aid of a theorem proving tool, attempting to prove or disprove the theorem. As Valmari observes, generally the proof uses several *invariants* and *variant functions*. An invariant is a property that all states of the system must have. Invariants are used for showing the system does only acceptable things. Variants are an upper limit to the number of times something can happen before something else happens. Variants are used to show the system makes progress, and eventually does the things it should do [194]. If a theorem proving tool is used, then the invariants and/or variant functions must be provided by the user. Such a tool also often requires much human assistance to prove the theorem. Hence, theorem proving methods require highly skilled personnel and can take a lot of time [53].

Other drawbacks of theorem proving methods are that they are focused on proving correctness and are difficult to use to provide *debugging information* (information on the nature and location of errors) and for the *analysis of behaviour* (i.e. answering the question of how the system behaves as opposed to whether it behaves in a given way) [194].

Conversely, SSMs are automatic and can be used for analysis and error detection as well as verification. State-based methods are therefore seen as one of the most promising formal reasoning techniques [194]. SSMs involve the investigation of all states that a system can reach given initial values for its input parameters. Most SSMs investigate the reachable states by constructing a directed graph called the *Reachability Graph* (RG) (also known as the *Occurrence Graph* [103]). In its most basic form a reachability graph (see Definition 7.1) is a directed graph consisting of all the states the system can reach from its given initial state. Each vertex of the graph is a unique state, and each directed edge is labelled with the action that leads to the next state (i.e. the next vertex of the graph). A reachability graph represents an *interleaving semantics* of a system. That is, it does not model the possibility of two or more actions occurring simultaneously. (It is possible to extend the reachability graph so that each edge stores a set of actions that occur simultaneously. Such a model is said to represent *step semantics*. However such extensions are rarely used.)

An example of a reachability graph is that of the dining philosophers Elementary Net system (Figure 2.5), shown in Figure 6.1. The top line of each vertex lists the philosophers that are thinking and the next line lists the ones that are eating. (In this figure we have not included transition labels with the edges of the graph.)

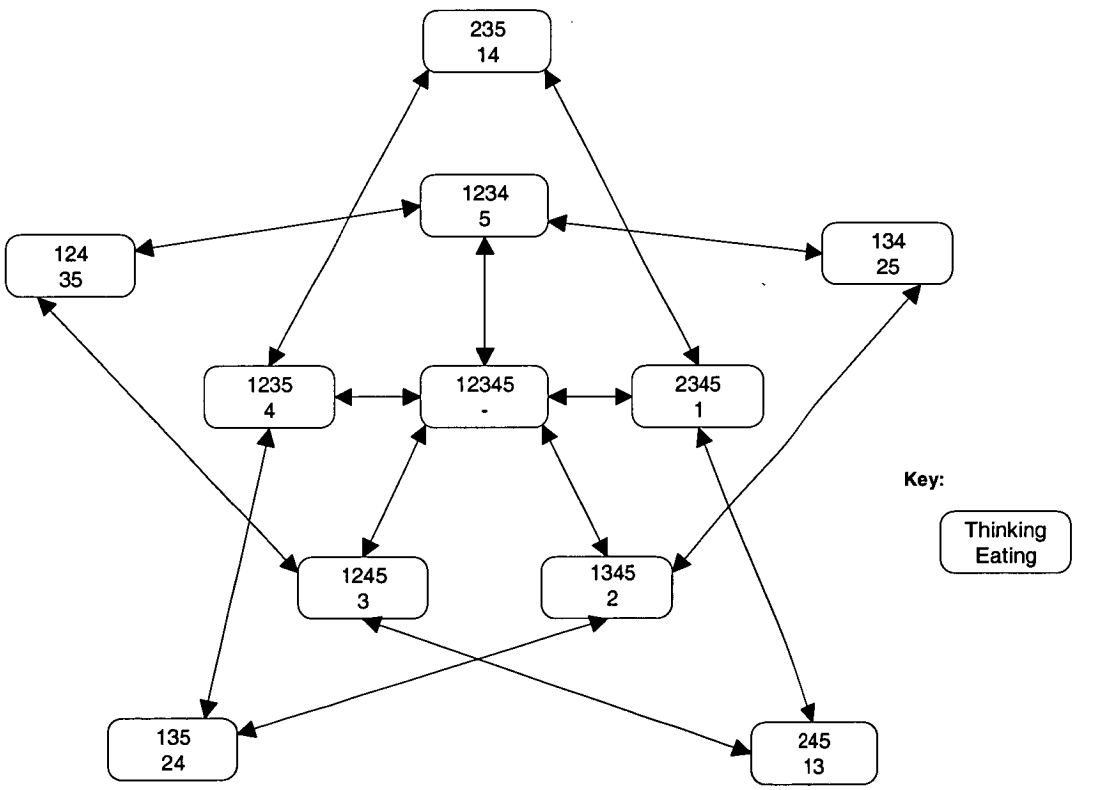


Figure 6.1: Dining philosophers reachability graph

SSMs can be applied by less trained personnel, and provided the state space is not too

large they are usually fast [194]. SSMs are often able to answer a wide range of analysis questions from the generation of one state space. If for some reason the system cannot be fully investigated then SSMs can be used to give partial answers. However, SSMs are not without their problems.

In general, SSMs are limited to investigating the system given a set of initial values. The state space must be regenerated if these initial values change. Another drawback of SSMs is that the system to be analysed must be finite-state, or a finite part of an infinite state system. (There have been methods for approximating certain types of infinite-state systems by finite-state systems, so that SSMs can be used [16, 8].)

The biggest problem associated with SSMs is the *state space explosion* problem. Unfortunately, owing to simple combinatorics, the number of states of a system tends to increase exponentially as the complexity of the system increases. That is, the size of a state space of a system tends to grow exponentially in the number of its processes and variables. The state space explosion means that the total number of states of a system is often far too large with respect to resources (time and space) to be fully generated.

The state space explosion problem is the primary obstacle to practical application of SSMs [194]. Holzmann illustrates that the state space explosion from composing even two simple communicating processes can easily make analysing a concurrent system impractical [93].

A general algorithm for computing the space of an arbitrary Petri Net is intuitively exponential. However unpleasant complexity results do not guarantee that state space techniques cannot be used in practice. The great advantages of SSMs have motivated many researchers to try to find ways of alleviating the problem. There is a large amount of literature on this area and we cannot hope to provide a comprehensive survey of all the techniques. Instead, in this chapter we introduce some of the strategies used (Section 6.2) and survey some of the reduction techniques (Section 6.3). For a more detailed examination we refer the reader to Valmari's excellent article [194].

6.2 State Space Reduction Strategies

There have been many attempts to combat state space explosion. Such attempts generally involve reducing the number of states constructed in the state space algorithm. A notable exception to this is the parallelisation of conventional reachability algorithms (e.g. Caselli [45]). The effectiveness of such parallel algorithms depends on factors such as the structure of the model and the architecture of the machine. In this section we consider general state space reduction strategies, including: removing information from the state space (Section 6.2.1), representing the state space more efficiently (Section 6.2.2), taking advantage of the compositional structure of a model (Section 6.2.3), preprocessing the models to produce a smaller state space (Section 6.2.4) or only investigating part of the whole state space (Section 6.2.5). The consideration of specific state space reduction methods is deferred to Section 6.3.

6.2.1 Removing Information

Fortunately, it is often the case that we are only interested in analysing certain properties of the model and can therefore reduce the information in the graph without affecting the

properties under consideration. The main obstacle with removing information is to remove it so that the properties of concern are not affected. A technique to reduce the information in the graph is *partial order reduction* (Section 6.3.1)

6.2.2 Compression

This approach involves storing the state space (or part of it) in a non-standard, dense way. The main obstacle with compressing the storage of the state space is that if the information is represented more densely, it can become too difficult to use. For example, the net itself is a representation of the state space but is too compressed to be useful. Compression techniques include those that take advantage of equivalence relations (Section 6.3.2) and binary decision diagrams (Section 6.3.3).

6.2.3 Compositional Techniques

The advantages of using a modular structure are well known, and therefore many formal methods advocate the use of modularity. (For some formal methods such as process algebraic techniques, compositionality is inherent). Compositional state space reduction techniques aim to take advantage of the modularity in a system to combat the state explosion. Compositional state space reduction is a desirable goal because it has the potential to significantly increase the size of the systems that can be analysed with given computer resources [191]. Such techniques (e.g. those of Christensen and Petrucci [51], Yeh and Young [206], Kemper [107], Cheung and Kramer [47], and Valmari [191, 192]) work by analysing each module separately and combining the results of these analyses to analyse the whole system. The approach taken by Yeh and Young [206] and by Cheung and Kramer [47] is a divide-and-conquer approach, where reachability graphs of subsystems are independently derived, simplified and then combined to form representations of successively larger parts of a complete system. The approach taken by Christensen and Petrucci [51] and Kemper [107] involves using the modular structure for partial order reduction (the Christensen and Petrucci [51] approach is discussed further in Section 6.3.5).

6.2.4 Preprocessing

The state space can be reduced by modifying the system description before the construction of the state space, or by taking the needs of the state space into account when the system is first modelled. For example, it is customary to use as few variables as possible and to restrict their type to be as small as possible. It is also customary to make the degree of atomicity of transitions as coarse as possible [194]. It is possible to use sound theories and automatic tools for preprocessing such that the modified model can be used to answer certain properties of the original model. An example of preprocessing is reduction theory which is discussed in Section 6.3.8.

6.2.5 Partial State Space Exploration

Some believe the state space explosion problem is so big and fundamental that it will never be possible to find the complete state space for large-scale systems. However, even partial analysis (that is, exploration of only part of the full state space) can detect errors that

tend to be fundamentally different from those found by simulation [194]. It is therefore common to only investigate critical parts of the whole state space, possibly by making assumptions or abstraction on the other parts of the state space.

Another common technique is to use *on-the-fly* verification. Here the algorithm that checks the validity of a property is integrated into the algorithm that constructs the state space. If at any stage in the construction an error against the property is found, then the algorithm stops. Hence the algorithm does not provide major advantages for a correct system, but can reduce the time to find errors in a system. This can be significant because incorrect systems tend to greatly increase the number of states compared to a correct system [194].

6.3 Survey of Efficient State Space Reduction Techniques

We now survey some of the more popular techniques that are used to combat state space explosion. This survey briefly explains the technique and discusses the time overhead incurred by the technique. We note that the amount of reduction achieved by each technique is problem specific and so general results on the amount of reduction achieved cannot be given.

6.3.1 Partial Order Reduction

The state space explosion is due, among other causes, to the modelling of concurrency by interleaving, or more accurately to the exploration of all possible interleavings of concurrent events. For instance the execution of n concurrent events is investigated by exploring all $n!$ interleavings of these events. Partial order reduction techniques use the fact that the total effect of a sequence of concurrent actions is independent of the order in which they are executed. They attempt to reduce the size of the reachability graph by considering (optimally) one interleaving of a sequence of concurrent actions.

Under partial order reduction, the full state space cannot be recovered. However, partial order reduction techniques ensure (provably so) that the reduced graph preserves the relevant properties of the full graph (e.g. various safety properties) so that reasoning can be performed on the reduced graph. The *Stubborn Sets* of Valmari [190], the *Persistent Sets* of Godefroid [79], and the *Ample Sets* of Peled [155] and combinations of these [79, 204] are partial order reduction techniques. These techniques differ on the actual details, but contain many similar ideas.

Stubborn Sets

A stubborn set [190, 194, 110] is the (restricted) set of transitions which are considered at a given state during reachability graph generation while maintaining the properties of interest. To build a reduced reachability graph, only transitions in the stubborn set are used to generate successor markings. The construction of stubborn sets depends on the properties being analysed or verified of the system, and on the type of Petri net that is being used.

Stubborn sets are a type of *persistent set* [79]. Intuitively, a subset S of the set of transitions enabled at a state M is called persistent if all transitions not in S that are enabled

in M, or in a state reachable from M through transitions not in S, are independent of all transitions in S. That is, all transitions interfering with a transition in S belong to S. This notion is captured by *dynamic* stubborn sets, as defined in Definition 6.1. This definition is from Valmari [194], but modified for Coloured Petri Nets.

Definition 6.1. A set $S \subseteq FE$ is *dynamically stubborn* at a marking $M \in \mathbb{M}_R$, if given $(t,c) \in S$, and $Y^* \in \sigma\mathbb{Y}$ such that all firing elements in Y^* are not in S then

a. $M[Y^*\rangle M_n \wedge M_n[(t,c)\rangle M'_n \Rightarrow \exists\, M' \in \mathbb{M}_R : (M[(t,c)\rangle M' \wedge M'[Y^*\rangle M'_n)$

b. There is at least one firing element $(t_k,c_k) \in S$ such that if $M[Y^*\rangle M_n$, then $M_n[(t_k,c_k)\rangle$.

Note:

a. All firing elements not in S that are enabled at M are independent of all firing elements in S.

b. If M is not a dead marking then S contains at least one firing element enabled at a marking reached from M.

The definition of dynamic stubbornness does not seem to imply a practical algorithm for computing dynamically stubborn sets. *Static* stubborn sets have been defined to provide a way for implementing stubborn sets. Static stubborn sets provide a sufficient static condition for a set being dynamically stubborn. That is if a set is statically stubborn then it is dynamically stubborn.

A stubborn set must be constructed for every marking encountered during reachability graph generation. Its construction must therefore be fast. An "optimal" stubborn set is one that is "minimal" in the sense it will no longer satisfy the stubborn set requirements if any transition is removed from it. The computation of the "optimal" (static) stubborn set typically takes quadratic time with respect to the number of transitions. Such complexity would make the stubborn set method infeasible for practical cases. Fortunately however, it is not necessary to produce optimal stubborn sets. The use of stubborn sets that are not optimal can lead to larger reduced graphs, but the total analysis time can be smaller than the time taken if the optimal stubborn sets are found. Valmari [190] has developed an algorithm that, in linear time (with respect to the number of transitions of the net), finds an often good stubborn set. This algorithm involves constructing a dependency graph where the vertices represent transitions and there is an edge from t_1 to t_2 if and only if t_2 is dependent on t_1 (i.e. t_1 enables or disables t_2). The stubborn set can be found from the dependency graph because if t_1 is in the stubborn set, an edge from t_1 to t_2 implies that t_2 is also in it.

A stubborn set for a Coloured Petri Net (CPN) can be constructed by unfolding it to an equivalent Place-Transition Net and using the method described above. However, unfolding can be expensive. By means of an example, Kristensen and Valmari [110] show that there are some CPNs for which a good stubborn set cannot be constructed without unfolding or doing something equally expensive. They propose a method that constructs a good stubborn set for CPNs without unfolding. This method involves constructing a dependency graph where the vertices represent classes of firing elements rather than individual firing elements. The size of such a dependency graph is intended to be proportional to the size of the CPN rather than its unfolded PTN. The firing element classes are deter-

mined by imposing extra structure on the CPN, namely process partitioning. If the CPN is partitioned into processes, then it is possible to perform stubborn set analysis without unfolding.

Sleep Sets

Sleep sets [79] are another partial order reduction technique. The sleep set method produces a reduced state space that is guaranteed to preserve all deadlocks of the full state space. As with the basic stubborn set method, a sleep set is associated with each state reached during the reachability graph construction. However, opposed to the stubborn set method, the sleep set is a list of transitions that are enabled in, but are not executed from, their associated state.

The sleep set method does not exploit information about the static structure of the net, as persistent algorithms do, but rather the past history of the reachability generation is used to detect transitions that are independent, but do not have to be explored. These transitions form the sleep set. The sleep set method can be illustrated using the net of Figure 6.2 (a) and its reachability graph shown in Figure 6.2 (b). In this net the transition t_1 is enabled and its occurrence will lead to the state s_1 as shown in the reachability graph. Similarly the transition t_2 is enabled and its occurrence will lead to the state s_2. Since the transitions t_1 and t_2 are independent then t_2 is still enabled after the occurrence of t_1 (i.e. t_2 is enabled in s_1) and t_1 is still enabled after the occurrence of t_2 (i.e. t_1 is enabled in s_2). However the exploration of both sequences t_1t_2 and t_2t_1 is wasteful since both these interleavings lead to the same state. In order to prevent this situation from occurring, the sleep set method does not explore t_1 in the state s_2. That is, t_1 is introduced in the sleep set associated with s_2.

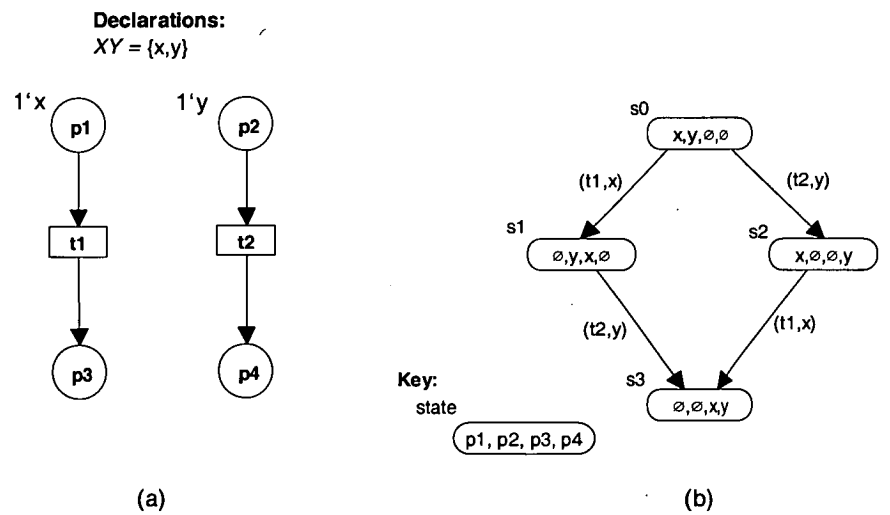


Figure 6.2: A simple CPN (a) and its reachability graph (b) for illustrating the sleep set method

The sleep set associated with the initial state, s_0, is empty. The sleep sets for the successors of a state s can be computed as follows. Let T be the set of transitions that have been selected to be explored from the state s. Suppose the first transition from T, $t_1 \in T$ leads to a state s'. The sleep set associated with the state s' is the subset of transitions in

the sleep set of s that are independent of t_1 in s. The sleep set of a second transition, t_2, taken from T at state s, is the subset of transitions of the sleep set of s that are independent of t_2 in s, augmented with t_1. In general, the sleep set associated with a state s' reached by a transition t from a state s is the subset of transitions in the sleep set associated with s that are independent of t in s, augmented with the transitions already taken from T.

Since transitions in the sleep set are independent, then the notion of sleep sets is orthogonal to the notion of persistent sets (including stubborn sets), and the algorithm presented by Godefroid [79] for finding sleep sets is integrated with the persistent set method.

6.3.2 Equivalence Reduction

A reachability graph with equivalence classes (RE-graph) is a graph of the state space, reduced in size by taking advantage of an equivalence relation for the set of token elements, and an equivalence relation for the set of firing elements. The RE-graph has a vertex for each equivalence class of markings that contains a reachable marking. It has an edge from vertex v_1 to v_2 for each equivalence class of firing elements that contain a firing element that leads from a marking of v_1 to a marking of v_2 [103]. Care must be taken to ensure that the RE-graph is consistent with the behaviour of the original net. Another drawback of the RE-graph approach is that the user must often specify the equivalence relations. Such relations are often not easy to find.

Symmetries

The *symmetry* technique — an equivalence reduction technique — was first proposed for Coloured Petri Nets by Huber et al [95]. Many concurrent systems are composed of components, the identities of which are immaterial or interchangeable. This kind of structural symmetry is reflected in the reachability graphs of such systems. The symmetry technique exploits such symmetries to reduce the number of markings considered during reachability graph generation. The idea has also been applied to other models of concurrency (e.g. [69, 54]).

A drawback of the symmetry method is that the symmetry relation must be given by the analyst (but its soundness can be automatically verified). An exception to this is the method of Chiola et al [48] which automatically determines the symmetries for models designed using a class of nets known as *Well-Formed Coloured Nets* [48]. In general however, the symmetry method requires external knowledge about the system so that the appropriate symmetry can be defined. A number of symmetries that are fairly easy for the analyst to detect in a model have been proposed. These include *identity symmetries*, *shift symmetries*, *permutation symmetries*, *related symmetries*, and *product symmetries*. Permutation symmetries, for example, can be obtained from permutations of the colours of the CPN (i.e. by bijective renamings of the colours).

Algorithms (e.g. [95, 181]) have been presented for constructing a reduced reachability graph that takes advantage of symmetries. The main disadvantage of these algorithms is that every generated marking must be compared for symmetry to every previously generated marking. On the other hand, this approach has the desirable property that no information is lost. The whole state space can be generated from the symmetric occurrence graph.

6.3.3 Binary Decision Diagrams

Each state of a state space is in fact a finite set. The domain of the set depends on the formalism, for example, for CPNs each state is a net marking. One way to represent a finite set is to use explicit enumeration. Suppose we have a set S over the domain 0..15, an enumeration of a set S might be $S = \{7, 11, 12, 13, 14, 15\}$. Represented in binary $S = \{0111, 1011, 1100, 1101, 1110, 1111\}$. A symbolic representation of S is $S = \{x \in \{0 \ldots 15\} \mid (x = 7 \ \vee \ x > 10)\}$. A symbolic binary representation is $S = \{v_1v_2v_3v_4 \in 0000 \ldots 1111 \mid (v_1 \wedge v2) \vee (((\overline{v_1} \wedge v_2) \vee (v_1 \wedge \overline{v_2})) \wedge v_3 \wedge v_4)\}$. Binary decision diagrams (BDDs) use a symbolic representation of the states and relations of the space rather than an explicit enumeration. Such a representation is sometimes more compact, and has proved particularly successful in analysing hardware. An example of a BDD given by Valmari [194] for the set S is given in Figure 6.3. The number $v_1v_2v_3v_4$ is in the set if and only if the path through the BDD ends with T, where the path is determined by selecting the output edge from each vertex v_i according to the value of v_i.

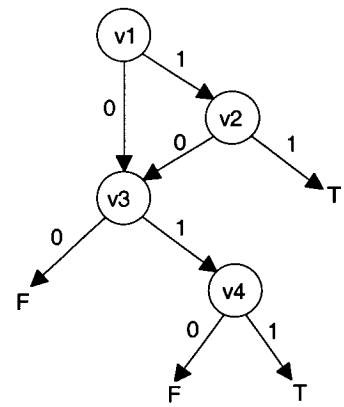


Figure 6.3: A BDD for $(v_1 \wedge v_2) \ \vee \ (v_3 \wedge v_4)$ [194, p. 496]

The BDD approach involves representing the transition relation by boolean formulas and storing these relations using BDDs [38]. This usually results in a much smaller representation of the relation, which means that extremely large state spaces can be checked, particularly in the domain of hardware verification. For example, [153] reports that spaces of size 10^{18} states can be efficiently calculated with a small BDD (10^3 vertices).

6.3.4 Holzmann's Bitstate Hashing

Another method for compressing the information in the state space is Holzmann's [94] bitstate hashing algorithm, also known as the *Supertrace* algorithm[1]. The algorithm works on the principle that the set of reachable states of a system is usually only a fragment of the syntactically possible states. For example, in the dining philosophers system with n philosophers, the state of a philosopher can be encoded in two bits and the state of a fork in one bit. There are $2^{3n} = 8^n$ syntactically possible states, but there are only $3^n - 1$ reachable states [194].

[1]The name *Supertrace* appears to come from the *Supertrace* tool, a successor to the *Trace* tool, developed by Holzmann for tracing protocol errors.

Most formalisms represent the state of the system as a bit vector of fixed length. A bit vector of length n has 2^n different values. It is common however that the set of reachable states of a system is only a small fragment of these values. The basic idea of the supertrace algorithm is to hash the original representation of the state to a shorter representation, and store the result in a hash table. In this approach, it is possible that there will be collisions, and therefore the algorithm may treat a newly found state as an already found state. The hashed approximation of the state space is not useful once the supertrace algorithm has terminated and so errors must be detected on-the-fly.

Thus Supertrace is an on-the-fly error detection method, *not* a verification method. That is, supertrace cannot be used to prove the absence of errors, but can be used to detect errors. The significant advantage of supertrace is that it can be used almost irrespective of the size of the state space of the system in question, and the amount of available memory. Supertrace always gives an answer within the resources given to it. The quality of the answer improves as the resources are increased.

6.3.5 Modular State Spaces

Modular State Space Analysis [51] constructs a set of reachability graphs, one for each module, together with a *Synchronisation Graph* that synchronises the module graphs. The result is that redundant interleavings due to interaction between the modules of the net are not considered. Modular Analysis can therefore be classed as a partial order reduction technique.

Modular analysis does not guarantee that the size of the state space will be reduced. There are pathological examples where there will be no reduction at all, for example a net consisting of only one module. However, modular analysis can be effective for well-designed systems. Christensen and Petrucci report results where the ordinary state space has 1 728 vertices and 7 368 edges, while the synchronisation graph has 4 vertices and 54 edges and the module graphs total 18 vertices and 18 edges [51].

Christensen and Petrucci [51] use modular nets, where modules interact by shared (fused) transitions. Each module reachability graph consists of markings that are reachable from the occurrence of transitions local to that module alone. The synchronisation graph represents the occurrence of fused transitions. The modular state space for the net of Figure 6.4 is given in Figure 6.5. The net of Figure 6.4 has two modules, module A and module B, with one fused transition, *Synch*. In Figure 6.5 the sets of vertices in dashed boxes are strongly connected components (SCCs). A given marking is represented in the synchronisation graph by the SCCs that the marking belongs to. That is, each vertex of the synchronisation graph is a product of SCCs of the modules. For example, the vertex of the synchronisation graph labelled by A_1B_1 represents the product of markings in the SCC $A1$ of the graph of module A with the markings in the SCC B_1. In this case the markings in the SCC A_1 is the set $\{a_1\}$ and the markings in the SCC B_1 is the set $\{b_1\}$, so A_1B_1 represents the marking a_1b_1.

The edges of the synchronisation graph represent the occurrence of fused transitions. To determine the successors of a vertex of the synchronisation graph, the occurrence of fused transitions are considered from all markings internally reachable from the vertex (i.e. all markings reachable by the occurrence of non-fused transitions). For example, in the net of Figure 6.4, to find the edges in the synchronisation graph from the vertex A_1B_1, the occurrence of *Synch* is considered from all markings internally reachable from A_1B_1.

That is, the occurrence of *Synch* is considered from a_1b_1, a_1b_2, a_1b_3, a_2b_1, a_2b_2, a_2b_3, a_3b_1, a_3b_2, a_3b_3. The only marking in which *Synch* is enabled is a_2b_2. The occurrence of the *Synch* transition at a_2b_2 results in the marking a_4b_4. The SCC representation of a_4b_4 is A_3B_3. Thus an edge is added to the synchronisation graph from A_1B_1 leading to A_3B_3.

The edges of the synchronisation graph additionally indicate the actual marking from which the fused transition occurs, and the marking resulting from its occurrence (i.e. the edge indicates the marking immediately preceding and following the fused transition). Thus the edge added from A_1B_1 to A_3B_3 indicated that the *Synch* transition occurs from the internal marking a_2b_2, leading to the marking a_4b_4. Labelling the actual markings on each edge of the synchronisation graph allows the graphs of modular state space to be combined to give the full reachability graph.

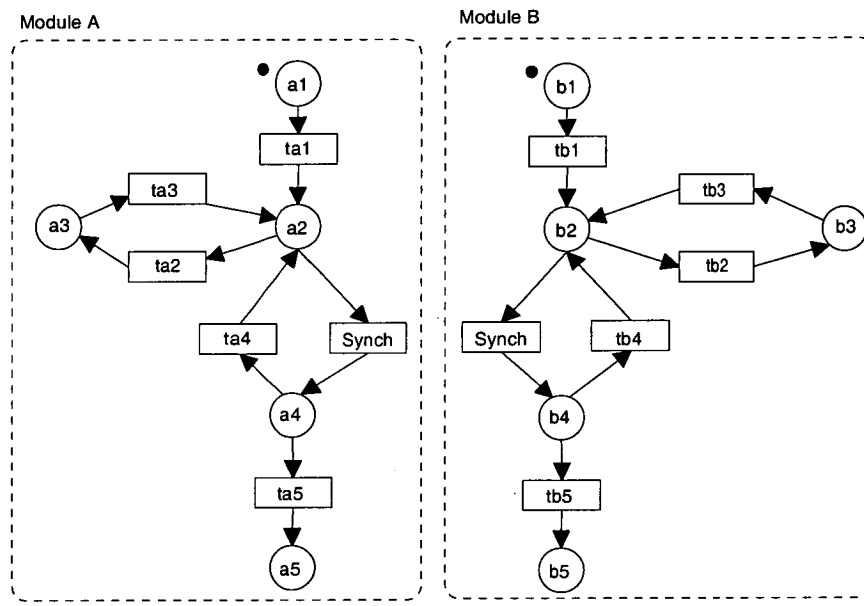


Figure 6.4: A modular Place/Transition net [51, p. 228]

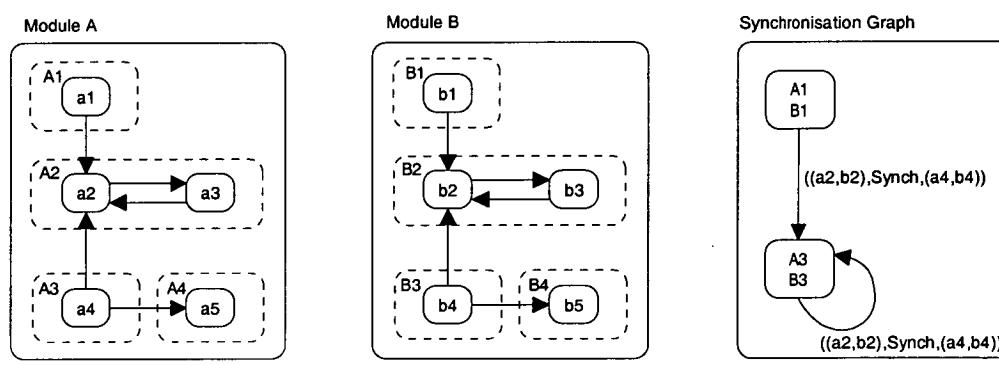


Figure 6.5: The modular state space of the net of Figure 6.4 [51, p. 229]

6.3.6 Parameterised Reachability Analysis

A method for constructing a reduced reachability graph by compressing the information stored in the graph has been proposed by Lindqvist [130]. This method is known as *Parameterised Reachability Analysis*. This technique involves the introduction of parameters (i.e. variables) to the marking of high level nets (Lindqvist presents the work in terms of Predicate Transition Nets). For example, the markings $M_1 = (a_1, b_1)$, $M_2 = (a_2, b_2)$ and $M_3 = (a_3, b_3)$ can be represented by the parameterised marking $M = (x_1, x_2)$, where x_1 and x_2 are parameters. Lindqvist formally defines parameterised markings and also presents a formulation of the transition rule for parameterised nets. With this method and definitions for determining when two parameterised markings are the same, Lindqvist is able to present an algorithm for constructing a parameterised reachability graph [130].

Unfortunately to fire a transition, parameters and constants must be matched (assigned) to the variables in the arc expressions. The best algorithm for this is little better than trying out every possible combination. Similarly, the comparison of the new marking with the already generated markings is computationally expensive. Also, given a parameterised reachability graph, in order to determine whether a particular marking is reachable from another (the relation represented by the standard reachability graph), it is necessary to "unfold" part of the reachability graph by fixing parameters.

It is unclear how practical the parameterising method is and whether the parameterised graph will yield significant reductions in size in industrial models. The parameterised method introduces significant overhead in developing the reduced reachability graph and extra work is required in the analysis of the graph.

6.3.7 Abstraction

In [55], Clarke et al propose a method for automatically constructing an abstract model of a given system and then analysing the abstract model. They report results of analysing a pipelined ALU circuit with over 10^{1300} states! Clarke et al express properties about the system using the propositional temporal logic CTL* [52]. In CTL* formulas are expressed using the standard operators of linear temporal logic and two path quantifiers, $\forall$ and $\exists$. $\forall$CTL* is a subset of CTL* in which only the $\forall$ quantifier is allowed. Clarke et al prove that the abstraction they use is *conservative* for properties expressed in $\forall$CTL*. That is, if a property expressed in $\forall$CTL* holds for the abstract model, then it holds for the actual model. However, if a property does not hold for the abstract model, then it may hold for the actual model, since a counterexample may be a behaviour of the abstract model but not of the actual model.

6.3.8 Reduction Theory

Reduction theory provides a set of rules that allow a net to be simplified while retaining some of the properties of the original net. This method is intended to reduce the number of reachable markings, and therefore simplify the analysis. The reduction rules are net transformations that can be applied based on only the structure and initial marking of the net. Berthelot [26, 27] presents several reduction rules for Place-Transition Nets that preserve a set of properties of the original net. These transformations involve fusing places, fusing transitions or fusing two nets together. In [86], Haddad generalises the most efficient

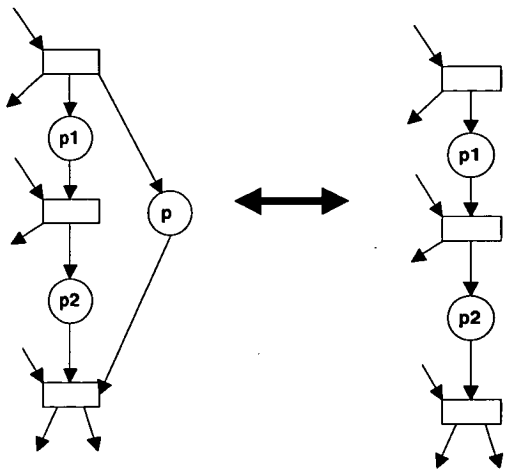


Figure 6.6: Simplification of an implicit place

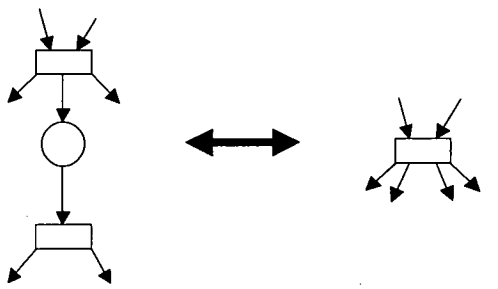


Figure 6.7: Post-fusion of transitions

reductions of Berthelot for CPNs. Figures 6.6 and 6.7 give examples of Berthelot's transformations. Figure 6.6 shows a transformation that removes a redundant place p because its marking is always sufficient to allow firing of its output transition. Figure 6.7 shows the fusion of two transitions.

The reduction technique is not always automatic because external knowledge may or may not be necessary depending upon the reduction rules that are applied.

It is worth noting that in some cases reduction can totally eliminate the need for reachability analysis. Berthelot has shown that bounded, live and persistent PTNs are fully reducible. That is, by applying a set of reduction rules it is possible to obtain a single transition that is trivially live. Unfortunately persistence is, in general, not satisfied. (A system is persistent if an enabled transition can only become disabled by firing).

An example of the use of net reductions is given in [174], where net reductions are applied to automatically generated Petri net models of Ada tasking in order to detect deadlock.

6.3.9 Combining Methods

It turns out that most of the methods are orthogonal, and when used together can often produce better results than when used in isolation.

Valmari [190] shows that the stubborn set method can be combined with the symmetry method (see Section 6.3.2). As stated by Jørgensen [106], for the dining philosopher net,

both the stubborn set method and the symmetry method reduce the reachability graph from exponential to quadratic in the number of philosophers. The combination of symmetries and stubborn sets reduces the reachability graph of the dining philosophers system from exponential to linear in the number of philosophers [190]. In [79], Godefroid shows that persistent sets (such as stubborn sets) can be combined with sleep sets.

In [186], Tiusanen also considers the combination of the symmetry and stubborn set methods, confirming that the two methods are indeed orthogonal. Tiusanen also suggests a way to combine such methods with symbolic model checking (employing binary decision diagrams).

In [50], it is suggested that Modular Analysis should be able to be combined with an equivalence technique (such as the symmetry technique). However the combination of Modular analysis and persistent sets is not obvious because they are both partial order reduction techniques.

6.3.10 Summary

Many techniques have been proposed to help alleviate the state explosion problem. To reduce the size of the state space it is necessary to make use of additional information such as the symmetry present in the net and the dependency between transitions.

More recent techniques such as Modular Analysis use the structure of the model to help reduce the state space. These approaches have the advantage that they do not require extra computation in each state. The work based on the modular structure of the models provides our inspiration for developing reachability algorithms that take advantage of the incremental structure present in many models.

Other lessons can also be learnt from examining other reachability techniques. We have observed that the data structures used in the algorithms for these techniques are vital to their performance. We have also observed that many of the previous state space reduction techniques are orthogonal, and can be combined to produce even greater reductions.

Table 6.1 presents a summary of the state space techniques surveyed, whether external knowledge is required, and the complexity of computation required in each state.

[2]The net must be Modular, but this is likely for a well designed model.

Technique	**Strategy**	**External Knowledge Required**	**Complexity of Calculations Required in each State**
Stubborn Sets	Removing information	No	Depends upon the algorithm. Linear in the number of transitions may give a non-optimal stubborn set.
Sleep Sets	Removing information	No	Possibly exponential
Symmetries	Compression	Yes	Possibly exponential
Binary Decision Diagrams	Compression	No	No extra work
Modular Analysis	Compositional	No^2	No extra work
Parameterised Analysis	Compression	No	Approximately exponential
Reduction Theory	Preprocessing	Possibly	All work is done prior to the generation of the reachability graph

Table 6.1: Summary of efficient state space techniques

Chapter 7

Incremental State Space Algorithms

We have developed algorithms aimed at using the forms of incremental change introduced in Part I of this thesis to help alleviate the state space explosion. We refer to these algorithms as *incremental algorithms*. In this chapter we present the standard state space algorithm (Section 7.1), and then we consider how it can be modified to cater for type, subnet and node refinement respectively (Sections 7.2 – 7.4). In Section 7.5 we present an algorithm that caters for a combination of type, subnet and node refinement. The incremental algorithms of this chapter have previously been presented [129].

To keep the following presentation as simple and clean as possible, we avoid detailed discussion of design choices and implementation details, preferring instead to defer these to Chapter 8. In particular, efficiency constraints imposed by the tool in which the algorithms were implemented (Maria [136]) mean that the implementation of the incremental algorithms are slightly different from the algorithms presented in this chapter. The differences and reasons for these differences are also discussed in Chapter 8. In Chapter 9 we examine the performance of the incremental algorithms compared to the standard algorithm.

7.1 The Standard State Space Algorithm

As was explained in Section 6.1 the standard way to investigate the state space is to develop a directed graph, called a *reachability graph*. A reachability graph of a CPN has a vertex (node) for every reachable state or marking of the system, and a directed edge (arc) for every possible transition that occurs with a given mode (see Definition 7.1). Definition 7.1 is adapted from that of Jensen [103, Def. 1.3]. As in the definitions of Jensen [103], we allow multiple edges between pairs of vertices.

Definition 7.1. A *reachability graph* (or *full reachability graph*) of a net $N = (P,T,A,C,E,\mathbb{M},\mathbb{Y},M_0)$ is a labelled directed graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$ where:

$\mathcal{V} = \mathbb{M}_R$, the set of vertices, each of which is a reachable marking.

$\mathcal{E} = \left\{ (M_1,(t,c),M_2) \in \mathcal{V} \times FE \times \mathcal{V} \;\middle|\; M_1[(t,c)\rangle M_2 \right\}$, the set of edges, each labelled by a firing element and $M_1, M_2 \in \mathbb{M}_R$.

The reachability graph of the net of Figure 7.1 (a) is shown in Figure 7.1 (b) (where the key indicates how the marking of each place contributes to a marking of the net). The reachability graph can be constructed by adding a vertex representing the initial marking (state) to the graph, and then adding edges and associated vertices to the graph for all the immediate successor markings. This process of adding immediate successors and edges is then repeated for each vertex for which immediate successors have not already been examined, until all reachable markings have been found.

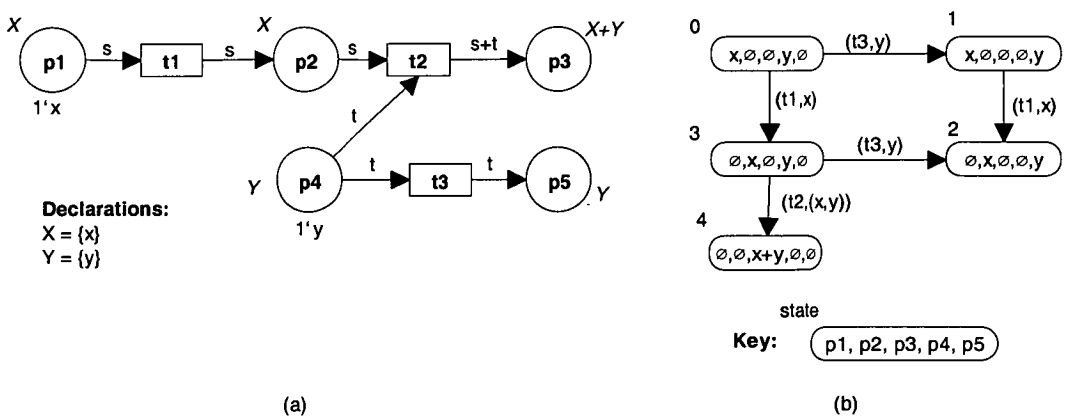


Figure 7.1: A simple net (a) and its reachability graph (b)

Algorithm 7.1 is the standard reachability graph algorithm for CPNs. It is based on that of Jensen [103, p. 5], and differs from it only to simplify the development of incremental versions. As discussed in Chapter 6, improvements such as partial order reduction, or equivalence based reduction can be made to this basic algorithm to help alleviate the state space explosion problem. The algorithm determines the reachability graph $\mathcal{G}$ for the net N starting from the marking M. The full reachability graph will be developed if $\mathcal{G}$ is initially empty, and M is equal to the initial marking of the net. However, we do not require that $\mathcal{G}$ is initially empty nor that M is the initial marking. This will be useful for the incremental algorithms described in Sections 7.2 – 7.5.

The function MATCH$(\mathcal{G},M)$ returns true if and only if M matches any vertex in $\mathcal{G}$. We do not indicate how matching of markings is performed. It could be something trivial like an equality test, or alternatively, something more subtle such as allowing for symmetry [103]. *Waiting* is a set of reachable markings, specifically those for which successors have not yet been examined. It is therefore initially set to $\emptyset$. Functions indicate the addition of vertices and edges to the graph. The function ADDVERTEX$(\mathcal{G},M)$ adds a vertex representing the marking M to the graph $\mathcal{G}$, while the function ADDEDGE$(\mathcal{G},(M_1,(t,c),M_2))$ adds an edge from M_1 to M_2 labelled by (t,c) to $\mathcal{G}$. The function SELECT(*Waiting*) returns a marking from the set *Waiting*. We do not indicate how the selection of this state is performed. The nature of this selection will determine whether the graph is constructed in a breadth-first manner, depth-first manner, or some other order.

The variable *possible* is a set of candidate transitions to be examined. We do not indicate how this set is calculated. In the worst case, it would be the set of all transitions. In the best case, it would be the set of transitions enabled at M. Unfortunately, there is no known heuristic that can efficiently determine exactly those transitions that are enabled. Therefore *possible* will normally include all enabled transitions plus others that are not enabled at M. The fewer disabled transitions included in the set, the better the performance of the algorithm.

The function EDGESFROM$(N,M_1,possible)$ returns the set of edges that result from the occurrence of a transition in the set *possible* at marking M_1, namely:

$$\left\{(M_1,(t,c),M_2) \;\middle|\; (t \in possible) \wedge M_1[(t,c)\rangle M_2\right\}$$

The function EDGESFROM therefore must determine which firing elements are enabled and this can be a bottleneck in the performance of the reachability graph algorithm. Pseudo code for the EDGESFROM function is also presented. The function ENABLEDFIRINGELEMENTS(N,M,t) returns the set of firing elements involving t enabled at marking M. ENABLEDFIRINGELEMENTS could be implemented using the *Transition Instance Analysis* algorithm [136, 137]. The Instance Analysis Algorithm is rather complicated, but the basic idea is to bind tokens in the input places, one at a time, to the variables on the input arcs of the transition being analysed.

Algorithm 7.1 Standard Reachability Graph Algorithm

```
REACHABILITYGRAPH(G,N,M)
begin
  Waiting := ∅
  if not MATCH(G,M) then
    ADDVERTEX(G,M)
    Waiting := {M}
  end if
  while Waiting ≠ ∅ do
    M1 := SELECT(Waiting)
    for all (M1,(t,c),M2) ∈ EDGESFROM(N,M1,possible) do
      if not MATCH(G,M2) then
        ADDVERTEX(G,M2)
        Waiting := Waiting + {M2}
      end if
      ADDEDGE(G,(M1,(t,c),M2))
    end for
    Waiting := Waiting − {M1}
  end while
  return G
end

EDGESFROM(N,M1,possible)
begin
  Result := ∅
  for all t ∈ possible do
    for all (t,c) ∈ ENABLEDFIRINGELEMENTS(N,M1,t) do
      M2 := M1 − E⁻((t,c)) + E⁺((t,c))
      Result := Result + {(M1,(t,c),M2)}
    end for
  end for
  return Result
end
```

The following sections present algorithms that cater for type, subnet, and node refinement respectively. The type and subnet algorithms use the basic property of a system morphism that if a step sequence is enabled at a given marking in the refined net then the corresponding abstract step sequence is enabled at the corresponding abstract marking of the abstract net (Definition 4.7 (c)).

7.2 Catering for Type Refinement

Given a net that has been derived from an abstract net by type refinement, we can use the reachability graph of the abstract net to help produce the reachability graph of the refined net. We refer to *reachability graphs*, *markings* and *firing elements* as either *abstract* or *refined*. In Section 7.1 we noted that a time consuming task in reachability graph generation is determining which firing elements are enabled for a given marking.

Recall that type refinement involves incorporating additional information in the net tokens and firing modes. Under type refinement the structure of the net remains the same and each token or mode of the refined net can be projected onto a token or mode respectively of the abstract net. Since type refinement is a system morphism (see Proposition 4.10), then the system morphism definition (Definition 4.7 (c)) requires the refined firing elements enabled at M project onto abstract firing elements enabled at $\phi(M)$. This means that, when determining which refined firing elements are enabled, we need only consider those that are derived from enabled abstract firing elements. Further, if neither the transition t nor its neighbouring places have been modified by type refinement, then the refined firing element (t,c) is enabled at marking M exactly when the corresponding abstract firing element (t,c) is enabled at the corresponding abstract marking $\phi(M)$. Finally, the follower marking M_1 can simply be determined by applying the changes to $\phi(M)$ to M.

We illustrate the above approach using the net of Figure 7.1 (a). Suppose that the type $X = \{x\}$ in this net is refined to $X = \{(x,1)\}$, and that the initial marking is changed so that the place p_1 contains the token $(x,1)$.

Given the marking $M_0 = (p_1,(x,1)) + (p_4,y)$ of the type refined net, then the corresponding marking in the abstract net is $\phi(M_0) = (p_1,x) + (p_4,y)$. From the reachability graph of the abstract net, we know that the firing elements (t_1,x) and (t_3,y) are enabled at $\phi(M_0)$. Since the places adjacent to t_1 have been type refined, we must check if firing elements involving t_1 are still enabled in the refined net. On the other hand the neighbouring places to t_3 have not changed, and so the enabled firing elements for t_3 in the refined net are exactly those that were enabled in the abstract net (i.e. (t_3,y)). The successor of firing (t_3,y) from $\phi(M_0)$ in the abstract net is $M_1{}' = (p_1,x) + (p_5,y)$. We can efficiently find the successor of M_0 with firing element (t_3,y) by applying the changes to M_0' to M_0. Hence the successor of M_0 by firing (t_3,y) is $M_1 = (p_1,(x,1)) + (p5,y)$.

Algorithm 7.2 modifies the EDGESFROM function of Algorithm 7.1 to take advantage of type refinement by only considering the firing elements enabled at $\phi(M)$ in the abstract net N' when determining the successors from M. To avoid confusion with functions of similar names in later algorithms we have appended "-TYPE" to the name of some functions. This means, for example, that the CHANGED-TYPE function is the CHANGED function for the algorithm that caters for type refinement. This convention is also used in the algorithms that cater for subnet refinement (functions are appended with "-SUBNET") and node refinement (functions are appended with "-NODE").

The ABSTRACTEDGESFROM function returns all the edges in the abstract net from the abstract marking $\phi(M)$. That is, it returns all the enabled abstract firing elements at $\phi(M)$ and the corresponding successor markings. We would usually expect the edges simply to be retrieved from the reachability graph of the abstract net, but they could be calculated on demand. The function CHANGED-TYPE(N,N',t) determines if the colour of neighbouring places of the transition t, or the transition itself, have been refined by type refinement. (It is necessary to consider the neighbouring places since it is possible that the

colour of the transition is not refined but the colour of a neighbouring place is refined. This could be achieved by changing functions on the arcs between the refined places and the transition.) The function UPDATE(N,N',M_1,M_2',t) takes as parameters the refined source marking, M_1, the abstract successor marking, M_2', and the transition that led to the abstract successor, t, and returns the refined successor marking. A simple way to implement this function is to replace in M_1 the marking of any places input and output to t with their marking from M_2'. (Note that since the transition t has not changed then its effect in the abstract net is the same as its effect in the refined net). The implementation of the UPDATE function is considered further in Section 8.5.2.

Algorithm 7.2 EDGESFROM modified to cater for type refinement

```
EDGESFROM-TYPE(N,N',M1,possible)
begin
  Result := ∅
  for all (φ(M1),(t,c'),M2') ∈ ABSTRACTEDGESFROM(N',φ(M1),φ(possible)) do
    if not CHANGED-TYPE(N,N',t) then
      M2 := UPDATE(N,N',M1,M2',t)
      Result := Result + {(M1,(t,c'),M2)}
    else
      for all (t,c) ∈ FE | φ((t,c)) = (t,c') do
        if M1 ≥ E⁻((t,c)) then
          M2 := M1 − E⁻((t,c)) + E⁺((t,c))
          Result := Result + {(M1,(t,c),M2)}
        end if
      end for
    end if
  end for
  return Result
end
```

7.3 Catering for Subnet Refinement

As was the case with type refinement, if a net is refined using subnet refinement, we can use the reachability graph of the abstract net to help determine the firing elements that are enabled at a given refined marking, and therefore reduce the time required to construct the reachability graph of the refined net.

In the case of subnet refinement, the system morphism, $\phi : N \to N'$, is a restriction of the net N. In other words, some of the components of N are ignored in N'. Since subnet refinement is a system morphism (see Proposition 4.12) then those refined firing elements enabled at M that are not ignored by the mapping will correspond to abstract firing elements enabled at $\phi(M)$. Thus, a refined firing element of N may map to the same abstract firing element of N', or may be ignored. To find the firing elements enabled at a given refined marking, M, we first consider the abstract firing elements enabled at the corresponding abstract marking, $\phi(M)$. Again, this would usually be done simply by using the reachability graph of the abstract net. The refined firing elements that are possibly

enabled at M are those refined elements that map to enabled abstract elements, together with those that are ignored in N'. So, if the refined transition t appears as it did in the abstract net, that is with no additional arcs or colours, then the enabling of firing element (t,c) in marking M is determined solely by the enabling of (t,c) at the abstract marking $\phi(M)$. If on the other hand the transition t has changed, either by being introduced in the refinement or by the addition of arcs or subnet refinement of its type or the type of neighbouring places, then we must check whether the refined firing element is enabled.

We illustrate the principle using the net of Figure 7.1 (a), modified by subnet refinement, as in Figure 7.2. Given the marking $M_0 = (p_1,x) + (p_4,y) + (p_6,z)$, the corresponding abstract marking is $\phi(M_0) = (p_1,x) + (p_4,y)$, where the marking of newly added places is ignored. The abstract firing elements enabled at $\phi(M_0)$ are (t_1,x), and (t_3,y). The transition t_1 has not been changed by the refinement, nor have its neighbouring places, and hence the refined firing element (t_1,x) is enabled in the refined net. Further, the abstract successor can be used to efficiently determine the refined successor. In this case, the successor of the initial marking in the abstract net is $M_1' = (p_2,x) + (p_4,y)$. The changes to places p_1 and p_2 must be applied to the initial marking of the refined net, namely M_0, giving $M_1 = (p_2,x) + (p_4,y) + (p_6,z)$.

On the other hand, the transition t_3 has been modified (it has an extra input arc), so, given that t_3 is enabled in the abstract net, then firing elements involving t_3 must be examined to determine whether they are enabled in the refined net. It turns out that the firing element $(t_3,(y,z))$ is enabled. Finally, there is a newly added transition, t_4, which must be examined to find out if it has any enabled firing elements.

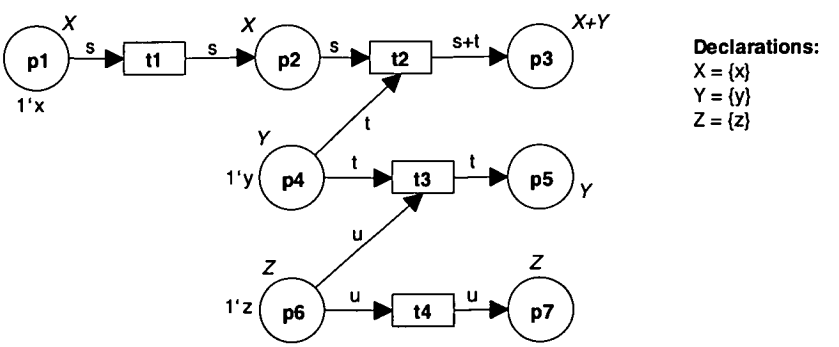


Figure 7.2: The net of Figure 7.1 refined using subnet refinement

Algorithm 7.3 modifies the EDGESFROM function of Algorithm 7.1 to cater for subnet refinement. The ABSTRACTEDGESFROM function is as described for Algorithm 7.2. It returns all the enabled abstract firing elements at $\phi(M)$ and the corresponding successor markings. We would usually expect the edges simply to be determined from the reachability graph of the abstract net, but they could be calculated on demand.

The function CHANGED-SUBNET(N,N',t) determines if the transition t or its neighbouring places have been modified by subnet refinement from their corresponding abstract version, or if the transition is added by subnet refinement. (It is necessary to consider the neighbouring places since it is possible that the colour of the transition is not refined but the colour of a neighbouring place is refined. This could be achieved by changing functions on the arcs between the refined places and the transition.) The function UPDATE(N,N',M_1,M_2',t) is as discussed in Section 7.2. The function MAPPED(t,ϕ) returns *true* if the transition t is mapped to a node in N' by the morphism ϕ (rather than being ignored).

Algorithm 7.3 EDGESFROM modified to cater for subnet refinement

```
EDGESFROM-SUBNET(N, N', M1, possible)
begin
  Result := ∅
  for all (φ(M1), (t,c'), M2') ∈ ABSTRACTEDGESFROM(N', φ(M1), φ(possible)) do
    if not CHANGED-SUBNET(N, N', t) then
      M2 := UPDATE(N, N', M1, M2', t)
      Result := Result + {(M1, (t,c'), M2)}
    else
      for all (t,c) ∈ FE | φ((t,c)) = (t,c') do
        if M1 ≥ E⁻((t,c)) then
          M2 := M1 − E⁻((t,c)) + E⁺((t,c))
          Result := Result + {(M1, (t,c), M2)}
        end if
      end for
    end if
  end for
  for all (t,c) ∈ FE | (t ∈ possible) ∧ not MAPPED(t, φ) do
    if M ≥ E⁻((t,c)) then
      M2 := M1 − E⁻((t,c)) + E⁺((t,c))
      Result := Result + {(M1, (t,c), M2)}
    end if
  end for
  return Result
end
```

7.4 Catering for Node refinement

For a net exhibiting node refinement, we develop the state space in a modular fashion, along the lines of Modular Analysis by Christensen and Petrucci [51]. This means that we generate a directed graph for each supernode and a directed graph to capture the global behaviour. This approach avoids much of the interleaving that would normally be present in the full reachability graph. Although this approach produces several reachability graphs (which was not the case for the type or subnet refinement), all three refinements can be combined and this is addressed in Section 7.5.

We refer to the collection of graphs as the *Refined-Node State Space* (RNSS) (Definition 7.16). The graph of a supernode is referred to as the *supernode graph* of that supernode. Each supernode graph captures only the local behaviour of its supernode. The graph that captures global behaviour is called the *global graph*. Essentially the global graph contains the abstract behaviour of the net.

In Sections 7.4.1 – 7.4.6 we first informally, and then formally, present the RNSS. In Section 7.4.8 we give an algorithm for constructing the RNSS. The graphs of the RNSS can be combined to obtain the full reachability graph, and this is considered in Section 7.4.7. In line with the terminology used for Modular Analysis [51], we refer to the process of combining the graphs as *unfolding* the RNSS. The various dynamic properties (reachability, dead markings, home properties, liveness, boundedness) can also be determined from the RNSS without unfolding. This is important since unfolding the RNSS may be an expensive operation. We consider determining the various dynamic properties without unfolding in Section 7.4.9. In Section 7.4.15 we consider an optimisation to the RNSS. Finally in Section 7.4.16 we compare the RNSS approach to that of Modular Analysis [51].

7.4.1 Informal Explanation of the RNSS

We explain the construction of the RNSS by considering the node refinement of Figure 7.1 (a) as in Figure 7.3. For each concept introduced in this section we indicate in parenthesis the section where formal definition(s) of that concept can be found, and then give an informal explanation of the concept.

It is useful to first introduce some terminology: we refer to a marking as *restricted* to a set of places, $X \subseteq P$, to mean the marking restricted to the token elements that involve a place from X (Definition 7.6). *Internal places (transitions)* are those places (transitions) local to a supernode; *external places (transitions)* are those not local to any supernode (Definition 7.2). *Terminal transitions* are those transitions internal to a supertransition, the occurrence of which results in an edge of the global graph. To allow the full reachability graph to be recovered the terminal transitions of a supertransition are defined to be the output border transitions of the supertransition (Definition 7.2). Later we consider an optimisation to the RNSS where the terminal transitions are redefined.

Each supernode graph of the RNSS contains only local information, namely reachable markings of the supernode and the associated enabled firing elements. (For technical reasons, the marking of the neighbouring places of a border transition of a supertransition are also included in its local reachability graph.) The vertices of the global graph refer to the *Strongly Connected Components* (SCCs)[1] (see Section 7.4.3) of the supernode

[1] Informally a SCC, S, of a directed graph is a set of vertices such that any vertex in S is reachable from any other vertex in S and S is not a subset of any larger such set.

graphs without specifying further details of the markings of the supernodes. Each edge of the global graph corresponds to the occurrence of an external or terminal transition from a marking reachable from a vertex of the global graph by a sequence of steps involving (only) transitions internal to the supernodes that can affect the enabling of the transition.

To construct the RNSS, we first add to the global graph the *Global Vertex* (Definition 7.7) representing the initial marking, M_0. In general a global vertex representing a marking M is the marking M with the marking of each supernode represented by its SCC index in the supernode graph. The RNSS after adding a global vertex representing M_0 is shown in Figure 7.4 (a). (To avoid cluttering the graphs, we have not included the labels on edges of supernode graphs.) Vertex v_0 of the global graph is the global vertex representing M_0. We now describe how a global vertex is calculated.

Consider a marking M. To find the global vertex representing M, for each supernode the marking M is restricted to the places of the supernode (including environment places if the supernode is a supertransition). If this restricted marking is not already present in the supernode graph then it is added to the supernode graph, the local state space of the supernode is developed, and the SCCs of the supernode graph are calculated. The global vertex corresponding to M is then the marking M restricted to external places, together with the SCC index of the marking M restricted to each supernode.

For example, to calculate the global vertex corresponding to M_0, first the marking of M_0 restricted to the places of the supertransition of t_3 (including environment places of the supertransition) is added to the supernode graph of t_3. In Figure 7.4 state 0 of the supernode graph of t_3 represents this restricted marking. Next the local state space of the supertransition is developed and SCCs are calculated — in Figure 7.4 state 0 of the supernode graph of t_3 is assigned the SCC index B_1. Similarly the marking M_0 restricted to the places of the superplace of p_2 is added to the supernode graph of p_2. In Figure 7.4 state 0 of the supernode graph of p_2 represents this restricted marking. The local state space of p_2 is developed (in this case there are no reachable markings from the state 0), and SCCs are calculated — in Figure 7.4 state 0 of the supernode graph of p_2 is assigned the SCC A_1. The global vertex representing M_0 is then the sum of: M_0 restricted to external places, (p_2, A_1), and (t_3, B_1).

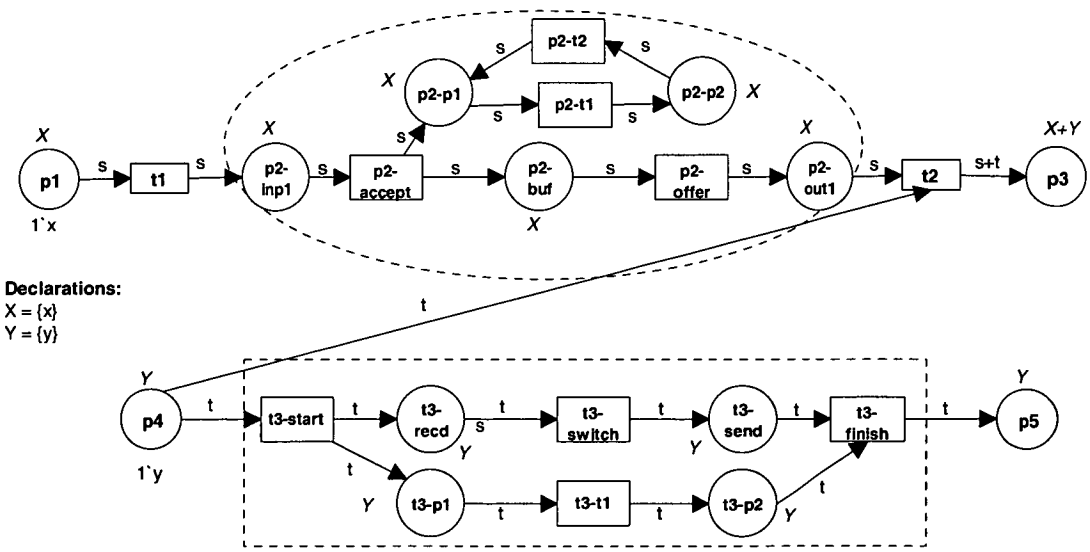
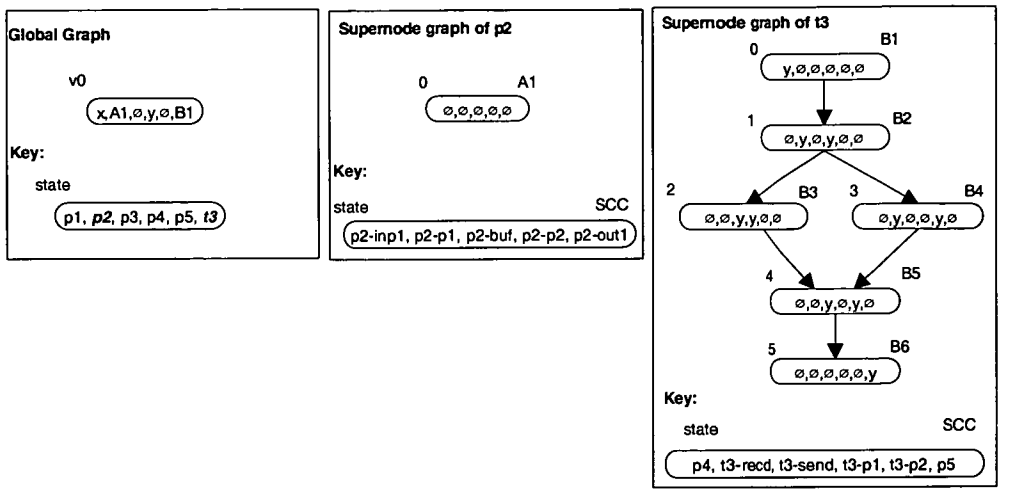
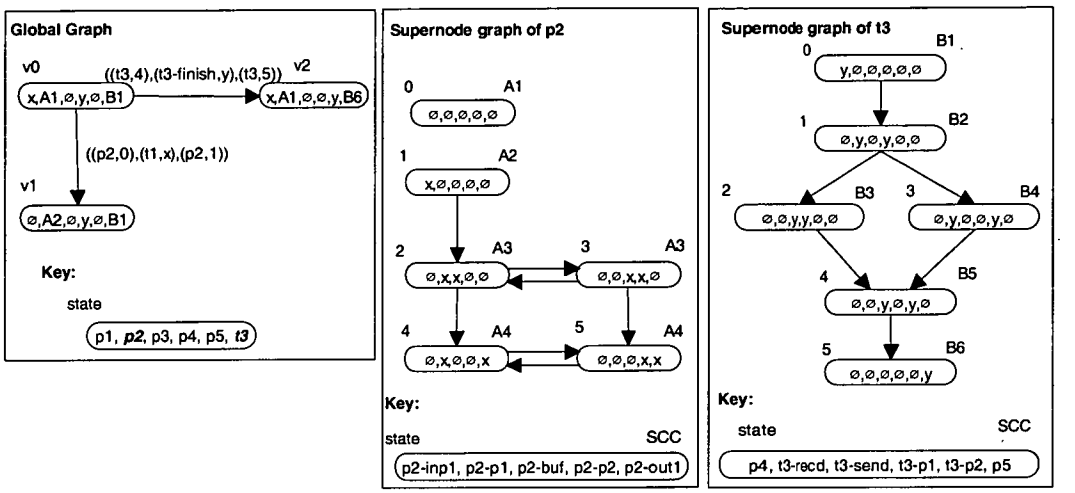


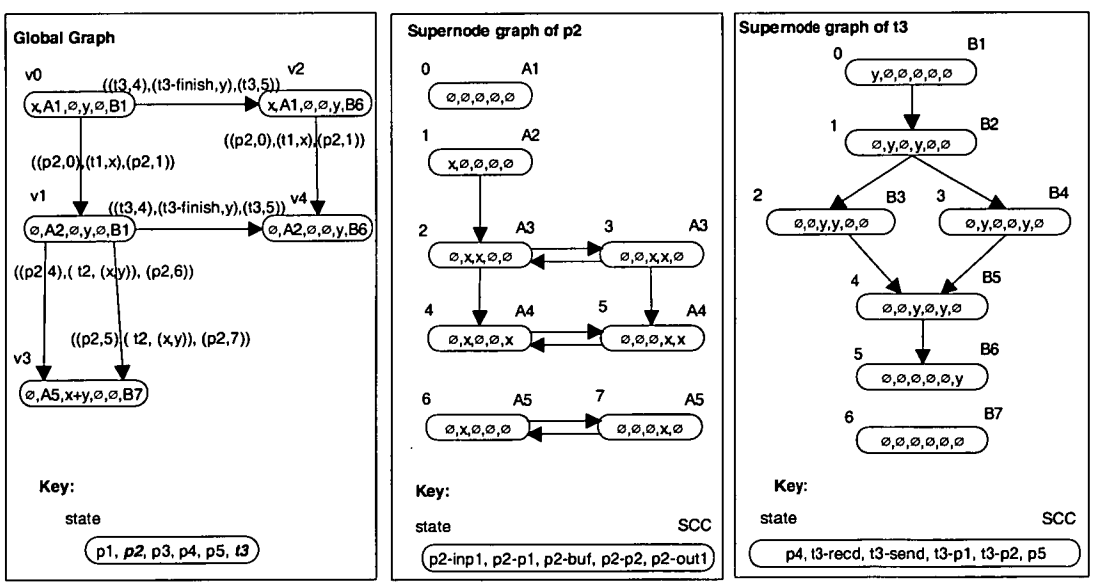
Figure 7.3: The net of Figure 7.1 refined using node refinement



(a)



(b)



(c)

Figure 7.4: RNSS generation for the net of Figure 7.3

The next step in the RNSS generation is to add immediate successors from the global vertex v_0 (which represents the marking M_0) to the global graph. That is, the next step is to add the (immediate) *Global Successors* (Definition 7.14) from the global vertex v_0. The result of adding (immediate) global successors from the global vertex v_0 is shown in Figure 7.4 (b). We now describe how the (immediate) global successors of a global vertex are found.

Consider a global vertex v representing a marking M. To find the global successors of v, we consider each external transition and then each supertransition. A global edge is added due to the occurrence of an external transition, t, from M or any markings reachable from M by transitions internal to superplaces input to t. (These are the internal transitions that can affect the enabling of t.) A global edge is added due to the occurrence of a terminal transition, t, of a supertransition from a marking reachable from M by transitions that are either internal to superplaces input to the supertransition or are internal to the supertransition. (Again, these are the internal transitions that can affect the enabling of t.)

Suppose the occurrence of the external or terminal transition, t, leads from the internally reached marking M_1 to M_2 (i.e. $M_1[(t,c))M_2$). Then the global edge added to the global graph leads from the vertex v to the global vertex representing the marking M_2. (The global vertex representing M_2 is calculated, and if it is not present in the global graph then it is added.) The edge added to global graph is labelled with the actual source marking (M_1), the firing element ((t,c)), and the actual successor marking (M_2). Labelling the edge with the actual markings (not just the relevant SCCs) allows the actual marking from which the transition occurred and the actual marking resulting from the occurrence of the transition to be recovered. Here we do not store the complete source (successor) marking but only the marking of places that are required to recover the actual source (successor) marking, given the source (successor) global vertex of the edge. Hence if the edge is due to an external transition then we store the source (successor) marking restricted to the places internal to those superplaces that are input to, or output from, the transition. If the edge is due to a terminal transition of a supertransition we store the source (successor) marking restricted to the places internal to those superplaces that are input to, or output from, the supertransition, together with the places internal to the supertransition.

Thus in our example, to add the global successors from the global vertex v_0 (representing M_0) we consider each external transition and then each supertransition. First consider the external transition t_1. There are no superplaces input to t_1 and so we only consider the occurrence of t_1 from M_0. The firing element (t_1,x) is enabled at M_0. The occurrence of this firing element leads to a marking $M_1 = (p_2\text{-}inp_1,x) + (p_4,y)$. Therefore there is a global edge from the global vertex v_0 to the global vertex representing M_1. Since there is no global vertex representing M_1 we add such a global vertex to the global graph. (As described above, this will result in the local state space of p_2 being developed.) In Figure 7.4 (b) the global vertex v_1 represents the marking M_1.

The edge from v_0 to v_1 is labelled with the source marking, M_0, restricted to the places of superplace of p_2, followed by (t_1,x), followed by the successor marking, M_1, restricted to the places of the superplace of p_2. In Figure 7.4 we use the state number of the superplace graph to represent the marking of the superplace. Thus the edge is labelled with $((p_2,0),(t_1,x),(p_2,1))$.

Next we consider the external transition t_2. Here there are no enabled transitions internal to the superplace of p_2 at M_0, and t_2 is not enabled at M_0, so there are no global successors from v_0 due to t_2. Finally we consider the supertransition of t_3. There are

no superplaces input to the supertransition of t_3, and so we must consider the occurrence of the terminal transition of the supertransition from those markings reachable from M_0 by transitions internal to the supertransition. That is we must consider the occurrence of t_3-*finish* from markings in the set $\{M_2, M_3, M_4, M_5\}$ where:

$$M_2 = (p_1,x) + (t_3\text{-}recd,y) + (t_3\text{-}p_1,y) \qquad M_3 = (p_1,x) + (t_3\text{-}recd,y) + (t_3\text{-}p_2,y)$$

$$M_4 = (p_1,x) + (t_3\text{-}send,y) + (t_3\text{-}p_1,y) \qquad M_5 = (p_1,x) + (t_3\text{-}send,y) + (t_3\text{-}p_2,y)$$

The transition t_3-*finish* is only enabled at M_5. (The supernode graph can be used to efficiently determine this). Here the firing element $(t_3\text{-}finish,y)$ leads to a marking $M_6 = (p_1,x) + (p_5,y)$. Hence we add an edge from v_0 to the global vertex representing M_6. Once again the global vertex representing M_6 must first be calculated. In Figure 7.4 the global vertex v_2 represents the marking M_6. The edge from v_0 to v_2 is labelled with the source marking, M_5, restricted to the marking of the supertransition of t_3, followed by the firing element, $(t_3\text{-}finish,y)$, followed by the successor marking, M_6, restricted to the marking of the supertransition of t_3.

The above process of adding successors is repeated for each global vertex for which immediate successors have not been examined. The complete RNSS is shown in Figure 7.4 (c). If we compare the ordinary state space of the whole system, with the RNSS, we observe that even for this trivial example, the RNSS is smaller than the ordinary state space. The RNSS contains a total of 20 vertices and 21 edges, while the ordinary state space contains 38 vertices and 88 edges.

7.4.2 Introduction to the Formal Definitions of the RNSS

The following sections give preliminary definitions which lead up to the RNSS definition. These sections define the concepts referred to in the informal explanation of the RNSS, and are presented in the order they were introduced in the informal explanation. Hence Section 7.4.3 considers strongly connected components, Section 7.4.4 considers global vertices, and Section 7.4.5 considers global successors. We are then able to define the RNSS in Section 7.4.6.

In all definitions in the remainder of this chapter we assume we have an abstract net N' and a refined net N related to N' by a system morphism $\phi : N \to N'$. Further to this we assume that each supernode of the net N does not contain nested supernodes. This means that each supernode graph is simply a directed graph (rather than a RNSS). We show in Proposition 7.23 that the RNSS can be unfolded to the full reachability graph. It follows from this that the definitions presented here can be modified so that a RNSS is used to represent the state space of each supernode, but we have not done so for clarity.

Before we begin with the preliminary definitions we recall some notation introduced in earlier chapters. First recall that FE represents the set of all firing elements of N. $FE|_X$ denotes the restriction of FE to firing elements involving transitions in $X \subseteq T$.

Next recall that, as defined in Definition 4.1, the *preimage* of an abstract node $x' \in P' \cup T'$ is: $N_{x'} = (P_{x'}, T_{x'}, A_{x'}, C_{x'}, E_{x'}, \mathbb{M}_{x'}, \mathbb{Y}_{x'}, M_{0x'})$. We use $P_{x'}$ to refer to the set of places of the preimage of x' and $T_{x'}$ to refer to the set of transitions of the preimage of x'.

Recall from Definition 4.2 that P'' is the set of abstract places that are replaced with a subnet in the refined net, where the subnet includes at least one transition; T'' is the set

of abstract transitions that are replaced with subnet in the refined net, where the subnet includes at least one place; and X'' as the set P'' together with the set T'', $X'' = P'' \cup T''$. We refer to the subnet $N_{p''}$ that replaces the abstract place $p'' \in P''$ in the refined net as the *superplace* of p'', and the subnet $N_{t''}$ that replaces the abstract transition $t'' \in T''$ as the *supertransition* of t''. The *supernode* of $x'' \in X''$ is either a superplace or supertransition.

Finally, recall from Definition 4.3 that for $x'' \in X''$ we refer to: the border input nodes of $N_{x''}$ by $inpbdr(N_{x''})$; the border output nodes of $N_{x''}$ by $outbdr(N_{x''})$; the border nodes of $N_{x''}$ by $bdr(N_{x''})$; the environment nodes input to nodes in $N_{x''}$ by $inpenv(N_{x''})$; the environment nodes output from nodes in $N_{x''}$ by $outenv(N_{x''})$; and the environment nodes input to, and output from, nodes in $N_{x''}$ by $env(N_{x''})$. In Definition 7.2 we define the function *terminal* to return the terminal transitions of a supernode. Initially the terminal transitions of a supernode are defined to be the output border transitions of the supernode. This definition allows the full reachability graph to be recovered from the RNSS. Later we consider an optimisation to the RNSS where the terminal transitions are redefined, and the full reachability graph cannot be recovered from the RNSS (see Section 7.4.15). Under certain conditions the optimised RNSS can be guaranteed to contain the same dead markings as the full reachability graph. In Definition 7.2 we also define the set of all *external transitions*, ET, the set of all *external places*, EP, the set of all *terminal transitions*, TT, and the set of all external and terminal transitions, ETT.

Definition 7.2. Given a marking $M \in \mathbb{M}$, then we denote:

a. $terminal(N_{x''}) = outbdr(N_{x''}) \cap T$

b. $ET = T - \bigcup_{x'' \in X''} T_{x''}$

c. $EP = P - \bigcup_{x'' \in X''} P_{x''}$

d. $TT = \bigcup_{x'' \in X''} terminal(N_{x''})$

e. $ETT = ET \cup TT$

Note:

a. $terminal(N_{x''})$ returns the transitions of the supernode of x'' that are output to the environment of the supernode. Only supertransitions can have terminal transitions.

b. ET is the set of *external transitions*. These are the transitions that are not local to any supernode.

c. EP is the set of *external places*. These are the places that are not local to any supernode.

d. TT is the set of *terminal transitions*. These are the terminal transitions of all supernodes.

e. ETT is the set of external and terminal transitions.

7.4.3 Strongly Connected Components

We now define SCCs and related concepts. The following definitions are derived from those of Jensen [103].

Definition 7.3. A *finite directed path* of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite sequence of vertices and edges $v_1 e_1 v_2 e_2 \cdots v_n$ where $v_i \in \mathcal{V}$ and $e_i \in \mathcal{E}$.

Definition 7.4. Two vertices $v_1, v_2 \in \mathcal{V}$ of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ are *strongly connected* if and only if there exists a finite directed path that starts in v_1 and ends in v_2, and a finite directed path that starts in v_2 and ends in v_1.

Definition 7.5. A *strongly connected component*, SCC, of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, is a directed graph $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$, where $\mathcal{V}_1 \subseteq \mathcal{V}$ is an equivalence class of strongly connected vertices, and $\mathcal{E}_1 \subseteq \mathcal{E}$ are all those edges where both the source and destination belong to $\mathcal{V}_1$.

We denote the set of all strongly connected components by *SCC*. As in [51] for a vertex $v \in \mathcal{V}$ and a component $c \in SCC$ we use the notation $v \in c$ to denote that v is one of the vertices in c. A similar notation is used for edges. For $v \in \mathcal{V}$, we use v^c to denote the strongly connected component to which v belongs.

7.4.4 Global Vertices

In this section we define the vertices added to the global graph. We refer to these vertices as *global vertices*. Each global vertex represents the marking of a supernode by its SCC index in its supernode graph. We first present notation for restricting a marking to a subset of places (Definition 7.6), and then using this notation we give notation for global vertices (Definitions 7.7 – 7.9).

Definition 7.6. Given a marking $M \in \mathbb{M}$, and a place $p \in P$ then M *restricted* to p is:

$$M|_p = \sum_{(p,c) \in M} (p,c)$$

We extend this to a set of places $X \subseteq P$:

$$M|_X = \sum_{x \in X} M|_x$$

We also extend this notation to a set of markings, $\mathcal{M} \subseteq \mathbb{M}$:

$$\mathcal{M}|_X = \bigcup_{M \in \mathcal{M}} \{M|_X\}$$

Further we give a shorthand notation for restricting the marking (or set of markings) M to the places in the superplace of $p'' \in P''$ as $M|_{p''} = M|_{P_{p''}}$ and for restricting the marking (or set of markings) M to the places in the supertransition of $t'' \in T''$, including the environment places of the supertransition of t'', as $M|_{t''} = M|_{(P_{t''} \cup env(N_{t''}))}$.

For the marking $M = (p_2\text{-}inp_1, x) + (p_4, y)$ of the refined net shown in Figure 7.3, the marking M restricted to the place p_4 is $M|_{p_4} = (p_4, y)$, the marking M restricted to the superplace of p_2 is $M|_{p_2} = (p_2\text{-}inp_1, x)$, and the marking M restricted to the supertransition of t_3 is $M|_{t_3} = (p_4, y)$.

We are now able to define global vertices. Recall that for a vertex v we use v^c to denote the strongly connected component to which v belongs. In a notation motivated by [51] we use $M^\natural$ to denote the global vertex corresponding to M, that is, the marking M with the marking of each supernode indicated by its SCC index in the corresponding supernode graph (Definition 7.7).

Definition 7.7. Given a marking $M \in \mathbb{M}$, then the *global vertex* corresponding to M is:

$$M^\natural = M|_{EP} + \sum_{x'' \in X''} (x'', (M|_{x''})^c)$$

This notation is also extended to a set $\mathcal{M} \subseteq \mathbb{M}$ of markings in the obvious way:

$$\mathcal{M}^\natural = \bigcup_{M \in \mathcal{M}} \{M^\natural\}$$

Going back to the example of Figure 7.3 and the associated RNSS shown in Figure 7.4 (c), consider the marking $M_1 = (p_2\text{-}p_1, x) + (p_3, x+y)$. We know that $M_1|_{p_2} = (p_2\text{-}p_1, x)$, and that $M_1|_{t_3} = \emptyset$. Now, according to the supernode graph of p_2, the SCC of $(p_2\text{-}p_1, x)$ is $A5$, and according to the supernode graph of t_3, the SCC of $\emptyset$ is $B7$. Therefore the global vertex corresponding to the marking M_1 is ${M_1}^\natural = (p_2, A5) + (p_3, x+y) + (t_3, B7)$. Similarly, for the marking $M_2 = (p_2\text{-}p_2, x) + (p_3, x+y)$ we have ${M_2}^\natural = (p_2, A5) + (p_3, x+y) + (t_3, B7)$. Note here that ${M_1}^\natural = {M_2}^\natural$.

Since a global vertex uses the SCC to represent each supernode marking, then as was the case in the above example, a number of markings of the net can be represented by the one global vertex. As in Definition 7.8, if v_1 is a global vertex we use ${v_1}^{-\natural}$ to denote the set of all markings that it represents.

Definition 7.8. For a global vertex v we define:

$$v^{-\natural} = \left\{ M \in \mathbb{M} \;\middle|\; M^\natural = v \right\}$$

For example, in the net of Figure 7.3 and the associated RNSS shown in Figure 7.4 (c), given $v = (p_2, A5) + (p_3, x+y) + (t_3, B7)$, then by examining the SCC $A5$ in the supernode graph of p_2 and the SCC $B7$ in the supernode graph of t_3 we establish that $v^{-\natural} = \{M_1, M_2\}$ where M_1 and M_2 are as described above.

As with markings, we can restrict a global vertex. Notation for this restriction is defined in Definition 7.9.

Definition 7.9. Given a marking $M \in \mathbb{M}$, a global vertex $v = M^\natural$, and a node $x \in EP \cup X''$ we define:

$$v|_x = \sum_{(x,c) \in v} (x, c)$$

We extend this to a set of places $X \subseteq EP \cup X''$:

$$v|_X = \sum_{x \in X} v|_x$$

7.4.5 Global Successors

As we discussed informally, the successors of a global vertex v are due to the occurrence of an external or terminal transition, t, from a marking internally reachable from a marking represented by v. Not all internally reachable markings are considered — only those markings internally reachable in a supernode that can affect the enabling of t are considered. The occurrence of t leads to a marking representing a global vertex, v_1. The edge from v to v_1 is labelled with a restriction of the actual source and successor markings to adjacent places.

In this section we first define internal steps (Definition 7.10). Notation for the markings internally reachable in a supernode that can affect the enabling of t is given in Definition 7.13, and global successors are defined in Definition 7.14. We end this section with a notation that allows us to restrict the actual source and successor marking to only those places that need to be stored (Definition 7.15).

Definition 7.10. Given a node $x'' \in X''$ we define the *internal steps* of the supernode of x'', denoted $\mathbb{Y}_{x''}$, as those steps involving only transitions internal to the supernode of x'', not including terminal transitions of the supernode of x'':

$$\mathbb{Y}_{x''} = \mu(FE|_S) \quad \text{,where } S = T_{x''} - terminal(N_{x''})$$

We extend this notation to a subset of nodes, $x \subseteq X''$ as:

$$\mathbb{Y}_X = \mu(FE|_S) \quad \text{,where } S = \bigcup_{x'' \in X} (T_{x''} - terminal(N_{x''}))$$

Definition 7.11. Given $M \in \mathbb{M}$ and $X \subseteq X''$ we define the function *internallyReachable* : $\mathbb{M} \times 2^{X''} \rightarrow 2^{\mathbb{M}}$ to return the set of markings reachable from M by a sequence of steps involving transitions internal to the supernodes of X:

$$internallyReachable(M,X) = \{M_1 \in \mathbb{M} \mid \exists\, Y^* \in \sigma\mathbb{Y}_X : M[Y^*\rangle M_1\}$$

Note that since $\sigma\mathbb{Y}_X$ includes the empty sequence then M is included in the set $internallyReachable(M,X)$. In a notation motivated by [51], we denote the set of markings internally reachable from M in X'' as:

$$[[M\rangle = internallyReachable(M,X'')$$

Further we generalise this notation for a set of markings $\mathcal{M} \subseteq \mathbb{M}$ such that:

$$[[\mathcal{M}\rangle = \bigcup_{M \in \mathcal{M}} [[M\rangle$$

This allows us to denote the set of markings internally reachable from a marking represented by the global vertex v by $[[v^{-t}\rangle$. This set is required to unfold the RNSS, and often used in the definitions and proofs that follow.

Definition 7.12. Given a transition $t \in ET \cup T''$ we define the function *inputSuperplaces* : $ET \cup T'' \rightarrow P''$ to return the set of abstract places whose corresponding superplace has input to t (if $t \in ET$) or input to the supertransition of t (if $t \in T''$):

a. if $t \in ET$ then:

$$inputSuperplaces(t) = \{p'' \in P'' \mid \exists\, p \in P_{p''} \,:\, p \in {}^\bullet t\}$$

b. if $t \in T''$ then:

$$inputSuperplaces(t) = \{p'' \in P'' \mid \exists\, p \in P_{p''} \,:\, p \in envinp(N_t)\}$$

Note:

a. If the transition is an external transition then $inputSuperplaces(t)$ returns the abstract places such that the superplace of the abstract place has input to t.

b. If the transition is an abstract transition then $inputSuperplaces(t)$ returns the abstract places such that the superplace of the abstract place has input to the supertransition of the abstract transition.

Definition 7.13. Given a marking $M \in \mathbb{M}$, and a transition $t \in ETT$, we define the markings internally reachable from M in supernodes that can affect the enabling of t, denoted $[_t[M\rangle$, as:

a. if $t \in ET$ then:

$$[_t[M\rangle = internallyReachable(M, inputSuperplaces(t))$$

b. if $\exists\, t'' \in T'' \,:\, t \in terminal(N_{t''})$ then:

$$[_t[M\rangle = internallyReachable(M, (inputSuperplaces(t'') \cup \{t''\}))$$

We extend this notation for a set of markings $\mathcal{M} \subseteq \mathbb{M}$:

$$[_t[\mathcal{M}\rangle = \bigcup_{M \in \mathcal{M}} [_t[M\rangle$$

Note:

a. If the transition is an external transition then $[_t[M\rangle$ is the set of markings internally reachable in superplaces input to t.

b. If the transition is a terminal transition of the supertransition of t'' then $[_t[M\rangle$ is the set of markings internally reachable in superplaces input to the supertransition of t'' together with the supertransition of t''.

As in Definition 7.14, we use $M_1[_t[(t,c)\rangle\rangle M_2$ to denote that M_2 is a global successor of M_1 by the firing element (t,c).

Definition 7.14. Marking M_2 is a *global successor* of M_1 due to a firing element $(t,c) \in FE|_{ETT}$, denoted $M_1[_t[(t,c)\rangle\rangle M_2$, if there exists $M_3 \in [_t[M_1\rangle$ such that $M_3[(t,c)\rangle M_2$.

We use $M_1[[\rangle\rangle M_2$ to denote that M_2 is a global successor of M_1 by any firing element from $FE|_{ETT}$, and we use $[[M\rangle\rangle$ to denote the set of all global successors from M by any firing element from $FE|_{ETT}$. That is $[[M\rangle\rangle$ is the closure of $M[[\rangle\rangle$.

Note: Marking M_2 is a global successor of M_1 by (t,c) if there is a marking M_3 internally reachable in the supernodes that affect the enabling of t, such that $M_3[(t,c)\rangle M_2$. We use $[[M\rangle\rangle$ to denote the set of global successors from M.

As we explained informally, each edge of the global graph is labelled with a restriction of the actual source and successor markings. If t is the transition of the edge, then as defined in Definition 7.15 we use $^\circ t^\circ$ to denote the places whose marking is stored with the global edge. The restriction notation (Definition 7.6) allows us to denote the source or successor marking, M, restricted to these places by $M|_{^\circ t^\circ}$.

Definition 7.15. Given a transition $t \in ETT$, we define the places whose marking is to be stored with each global edge as:

a. if $t \in ET$ then:

$$^\circ t^\circ = \bigcup_{p'' \in P''} \{p \in P_{p''} \mid \exists p_1 \in P_{p''} : p_1 \in {}^\bullet t \cup t^\bullet\}$$

b. if $\exists\, t'' \in T'' : t \in terminal(N_{t''})$ then:

$$^\circ t^\circ = P_{t''} \cup env(N_{t''}) \cup \bigcup_{p'' \in P''} \{p \in P_{p''} \mid \exists p_1 \in P_{p''} : p_1 \in env(N_{t''})\}$$

Note:

a. If the transition is an external transition then we must store the marking of each place internal to a superplace input to, or output from, t

b. If the transition is a terminal transition of a supertransition, then we must store the marking of each place internal to the supertransition, together with the marking of each place in the environment of the supertransition, together with the marking of each place internal to a superplace input to, or output from, the supertransition.

7.4.6 RNSS Definition

Finally, we can now define the RNSS (Definition 7.16). Explanatory notes follow this definition. They are intended to be read in parallel with the definition.

Definition 7.16 (The Refined Node State Space (RNSS)). We define the Refined Node State Space as $RNSS = (\mathcal{G}, \mathcal{G}_{X''})$ where:

a. $\mathcal{G} = (\mathcal{V}_{\mathcal{G}}, \mathcal{E}_{\mathcal{G}})$ is a directed graph, called the *global graph*, where:

1. $\mathcal{V}_{\mathcal{G}} = [[M_0\rangle\rangle^{\phi} \cup \{M_0{}^{\phi}\}$

2. $\mathcal{E}_{\mathcal{G}} = \left\{(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{V}_{\mathcal{G}} \times (\mathbb{M} \times FE|_{ETT} \times \mathbb{M}) \times \mathcal{V}_{\mathcal{G}}\right\}$

where $M_1 \in [_t[v_1{}^{-\phi}\rangle$, $M_2 \in v_2^{-\phi}$, $m_1 = M_1|_{^\circ t^\circ}$, $m_2 = M_2|_{^\circ t^\circ}$, and $M_1[(t,c)\rangle M_2$.

b. $\mathcal{G}_{X''} = \{\mathcal{G}_{x''} \mid x'' \in X''\}$ is a set of state spaces, one for each supernode, such that: $\mathcal{G}_{x''} = (\mathcal{V}_{x''}, \mathcal{E}_{x''})$ where:

1. $\mathcal{V}_{x''} = \bigcup_{v \in \mathcal{V}_{\mathcal{G}}} \left([[v^{-\phi}\rangle\rangle\right)\big|_{x''}$

2. $\mathcal{E}_{x''} = \left\{(m_1,(t,c),m_2) \in \mathcal{V}_{x''} \times FE|_{(T_{x''}-terminal(N_{x''}))} \times \mathcal{V}_{x''} \;\middle|\; m_1[(t,c)\rangle m_2\right\}$

Note:

a. The vertices of the global graph are the global vertices representing all global successors from the initial marking (i.e. all markings reachable according to Definition 7.14) together with the initial marking. The edges of the global graph are due to the occurrence of a firing element involving an external or terminal transition, t, from a source marking, M_1. The marking M_1 is internally reachable in the supernodes that can affect the enabling of t from a marking represented by a global vertex, v_1 (i.e. $M_1 \in [_t[v_1{}^{-\phi}\rangle$). The successor marking, M_2 is represented by a global vertex (i.e. $M_2 \in v_2^{-\phi}$) and the markings stored with the edge are the source and successor marking restricted to $^\circ t^\circ$. Note that the markings stored with each global edge could represent the state of the supernode using its state number from the supernode graph, as is done in Figure 7.4 and in the implementation (see Chapter 8). We have omitted this from the definition since it is a storage issue.

b. Every supernode has an associated graph. The vertices of a supernode graph contain the restriction to the supernode of every marking internally reachable from a global vertex. (Note that since a supertransition can have input from a superplace, then it is necessary to consider the markings internally reachable in all adjacent supernodes not just the markings internally reachable in the supernode of x''.) There is an edge of the supernode graph for every transition internal to the supernode that is enabled at a vertex of the supernode graph (not including terminal transitions). That is, the edges of the supernode graph correspond to all enabled transitions internal to the supernode (not including terminal transitions).

7.4.7 Unfolding the RNSS

In this section we define how the RNSS graphs can be combined into the full reachability graph. In line with the terminology used by Christensen and Petrucci [51], we refer to the process of combining the graphs of the RNSS as *unfolding* the RNSS. In Definition 7.17 we define the unfolded RNSS to be a directed graph. The vertices and edges of the unfolded RNSS are based on the markings internally reachable from global vertices (i.e. the markings in $[[v^{-t}\rangle$ where v is a global vertex). In Definition 7.11 the set $[[v^{-t}\rangle$ is defined in terms of the *internallyReachable* function. After the definition of the unfolded RNSS we consider how the supernode graphs of the RNSS can be used to implement the *internallyReachable* function and give an algorithm for this function (Algorithm 7.4). This algorithm therefore tells us how the unfolded RNSS can be constructed from the RNSS. In the last part of this section we prove that the unfolded RNSS is isomorphic to the full reachability graph (Proposition 7.23).

Definition 7.17. Given a RNSS, $(\mathcal{G}, \mathcal{G}_{X''})$ then the *unfolded RNSS* is the graph $RG = (\mathcal{V}_u, \mathcal{E}_u)$ where:

1. $\mathcal{V}_u = \bigcup_{v \in \mathcal{V}_\mathcal{G}} [[v^{-t}\rangle$

2. $\mathcal{E}_u = \mathcal{E}_1 \cup \mathcal{E}_2$ where:

$$\mathcal{E}_1 = \bigcup_{(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_\mathcal{G}} \left\{(M_1,(t,c),M_2)\right\}$$

where $M_1 \in [[v_1^{-t}\rangle$, $M_2 \in [[v_2^{-t}\rangle$, $M_1|_{{}^\circ t^\circ} = m_1$, $M_2|_{{}^\circ t^\circ} = m_2$, and $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$

$$\mathcal{E}_2 = \bigcup_{(m_1,(t,c),m_2) \in \mathcal{E}_{x''}} \{(M_1,(t,c),M_2)\}$$

where $x'' \in X''$, $M_1, M_2 \in \mathcal{V}_u$, $M_1|_{x''} = m_1$, and $M_2 = M_1 - m_1 + m_2$

Note: The vertices of the unfolded RNSS consist of all markings internally reachable from a marking corresponding to a global vertex. (As described in Algorithm 7.4, these internally reachable markings can be found using the supernode graphs.)
The edges of the unfolded graph can be split into two sets: $\mathcal{E}_1$ and $\mathcal{E}_2$. The set $\mathcal{E}_1$ contains edges due to the occurrence of an external or terminal transition. It consists of edges $(M_1,(t,c),M_2)$ for each edge $(v_1,(m_1,(t,c),m_2),v_2)$ in the global graph, where M_1 is internally reachable from the global vertex v_1, and M_2 is internally reachable from the global vertex v_2. Here it is required that M_1 restricted to the places whose marking is stored with the edge (i.e. $M_1|_{{}^\circ t^\circ}$) is equal to the source marking stored with the edge (m_1), and similarly the M_2 restricted to the places whose marking is stored with the edge (i.e. $M_2|_{{}^\circ t^\circ}$) is equal to the successor marking stored with the edge (m_2). Further it is required that M_1 restricted to the places whose marking is not stored with the edge is equal to M_2 restricted to the places whose marking is not stored with the edge (i.e. $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$). The set $\mathcal{E}_2$ consists of all edges due to the occurrence of an internal transition.

Note that for the set $\mathcal{E}_1$ of Definition 7.17 we have $M_1 \in [[v_1^{-t}\rangle$ rather than $M_1 \in [_t[v_1^{-t}\rangle$ (as is the case in the RNSS definition, Definition 7.16). Similarly we have $M_2 \in [[v_2^{-t}\rangle$ rather than $M_2 \in v_2^{-t}$. This allows for the fact that M_1 can be reached from a marking in v_1^{-t} by activity internal to supernodes which are not adjacent to t. The effect of this internal activity needs to be retained in the follower marking.

The unfolded RNSS requires the set of markings internally reachable from the markings represented by a global vertex. That is it requires the set $[[v^{-t}\rangle$ where v is a global vertex. Suppose we have a marking M such that $M^t = v$. Since for each supernode, the supernode marking of all markings in v^{-t} are in the same SCC then the set of markings internally reachable from the markings represented by v is equal to the markings internally reachable from M. That is $[[v^{-t}\rangle = [[M\rangle$.

The function FINDANINVERSE of Algorithm 7.4 uses the supernode graphs to construct a marking M such that $M^t = v$. Firstly M is set to v restricted to the marking of external places ($M := v|_{EP}$). Then for each supernode $x'' \in X''$, a marking of the supernode, m, is added to M. The supernode marking m is a vertex of the supernode graph of x'' and is such that the SCC of m is the same as that stored by v for that supernode. From Definitions 7.7 and 7.9 it follows that $M^t = v$.

Therefore to find the set $[[v^{-t}\rangle$ we can first use the function FINDANINVERSE to find a marking M such that $M^t = v$, and then find the set $[[M\rangle$. In Definition 7.11 the set $[[M\rangle$ is defined as $internallyReachable(M,X'')$. We now consider how the supernode graphs of the RNSS can be used to implement the $internallyReachable(M,X)$ function and give an algorithm for this function (Algorithm 7.4). The basic idea of the algorithm is to first select a node $x \in X$, and to use the supernode graph of x to construct all the internally reachable markings from M in the supernode of x. Then for each of those internally reachable markings to construct the internally reachable markings in the supernode of another node from X, and so on until all nodes in X have been considered. However, since a supertransition may have input from a superplace, then we must order the supernodes so that before the internally reachable markings of a supertransition are considered, the internally reachable markings of all superplaces input to the supertransition have been considered. (Considering all superplaces, and then all supertransitions would suffice.) The function *order*, defined in Definition 7.18, returns a suitably ordered set of supernodes. (Recall from Definition 2.1 that the set of all permutations of a nonempty set A is denoted $\pi(A)$.)

Definition 7.18. Given a set of abstract nodes that are refined, $X \subseteq X''$, we define the function $order : 2^{X''} \to 2^{X''}$ to return a permutation of X, where each node representing a superplace appears in the set before any node representing supertransitions that have input from the superplace. That is: $order(X)$ is a set $\{x_1'', x_2'', \ldots, x_n''\} \in \pi(X)$ such that $\forall i,j \in \{1,\ldots,n\} : (x_i'' \in T'') \wedge (x_j'' \in inputSuperplaces(x_i'')) \Rightarrow i > j$.

Algorithm 7.4 is an algorithm for the INTERNALLYREACHABLE function. The function ORDER is as defined in Definition 7.18. The function LOCALLYREACHABLE($\mathcal{M}$,x_i'') returns the markings reachable from any marking in the set $\mathcal{M}$ by transitions local to the supernode of x_i''. These markings can be found from the supernode graph of x_i''.

The INTERNALLYREACHABLE function of Algorithm 7.4 may appear to have a multiplicative effect, since the LOCALLYREACHABLE($\mathcal{M}$,x_i''), function finds reachable markings from every marking already considered (i.e. markings in $\mathcal{M}$). However, in the implementation of the LOCALLYREACHABLE($\mathcal{M}$,x_i'') function it is only necessary to examine the supernode graph x_i'' for markings from $\mathcal{M}$ where the local marking of the supernode of x_i'' is distinct. That is, in the implementation of LOCALLYREACHABLE($\mathcal{M}$,x_i'') function it is only necessary to examine the supernode graph of x_i'' for markings from the set $\mathcal{M}|_{x_i''}$. In most cases the supernode graphs do not interact[2], and so commonly there will be relatively few distinct markings. Further to this, we do not expect that the INTERNALLYREACHABLE algorithm will be used to unfold the RNSS in practice. Instead we expect the various dynamic properties to be determined directly from the RNSS (see Section 7.4.9).

Algorithm 7.4 FINDANINVERSE(v) and INTERNALLYREACHABLE(M,X)

FINDANINVERSE($\mathcal{G}_{X''}$,v)
begin
  $M|_{EP} := v|_{EP}$
  for all $x'' \in X''$ **do**
    select $m \in \mathcal{V}_{x''}$ such that $(x'',m^c) = v|_{x''}$
    $M := M + m$
  end for
  return M
end

INTERNALLYREACHABLE(M,X)
begin
  $\mathcal{M} := \{M\}$
  $\{x_1'',x_2'',\ldots,x_n''\} :=$ ORDER(X)
  $i := 1$
  while $i \leq n$ **do**
    $\mathcal{M} := \mathcal{M} \cup$ LOCALLYREACHABLE($\mathcal{M}$,x_i'')
    $i := i + 1$
  end while
  return $\mathcal{M}$
end

[2]Only if a supertransition has input from a superplace will their graphs interact.

We now prove that Algorithm 7.4 finds the set of markings internally reachable from M in X (i.e. that it finds *internallyReachable*(M,X)). First we define an *ordered internal sequence* (Definition 7.19). This is a sequence of internal steps (Definition 7.10) where the internal activity of each supernode appears in a given order.

Definition 7.19. Given a set of refined nodes $X = \{x_1'', \ldots, x_n''\} \subseteq X''$, we define the function *orderedInternalSeq* : $2^{X''} \rightarrow 2^{\sigma\mathbb{Y}_X}$ to return the sequences from $\sigma\mathbb{Y}_X$, where the internal activity or each supernode appears in the same order as the nodes appear in X. That is: *orderedInternalSeq*$(\{x_1'', \ldots, x_n''\}) =$

$$\left\{ Y_{x_1''}{}^* \cdots Y_{x_n''}{}^* \in \sigma\mathbb{Y}_X \mid \forall i \in \{1,\ldots,n\} \; : \; Y_{x_i''}{}^* \in \sigma\mathbb{Y}_{x_i''} \right\}$$

Clearly Algorithm 7.4 constructs the set of markings reachable from M by any sequence in *orderedInternalSeq*(*order*(X)). In Proposition 7.20 we prove that a marking is internally reachable from M in X if and only if it is reachable from M by a sequence from the set *orderedInternalSeq*(*order*(X)). It follows immediately that Algorithm 7.4 constructs the set *internallyReachable*(M,X).

Proposition 7.20. Given a global vertex $v \in \mathcal{V}_G$ of a RNSS, a marking $M \in \mathbb{M}$ such that $M^{\ell} = v$, a set of nodes $X \subseteq X''$, and an ordered permutation of X, $X_1 = order(X) = \{x_1'', \ldots, x_n''\}$, then:

$$M_1 \in internallyReachable(M,X) \Leftrightarrow \exists\, Y^* \in orderedInternalSeq(X_1) \; : \; M[Y^*\rangle M_1$$

Proof.
($\Rightarrow$) Suppose $M_1 \in$ *internallyReachable*(M,X). Then by Definition 7.11 there exists $Y^* \in \sigma\mathbb{Y}_{X''}$ such that $M[Y^*\rangle M_1$. Suppose we reorder Y^* to a sequence $Y_1{}^* = Y_{x_1''}{}^* \cdots Y_{x_n''}{}^*$ such that each $Y_{x_i''}{}^*$ contains only internal activity of the supernode of x_i'', and the ordering of $Y_1{}^*$ matches that of X_1. Thus $Y_1{}^* \in$ *orderedInternalSeq*(X_1).

We now show that this reordering does not affect the enabling of the sequence. Since no external or terminal transition occurs in $Y_1{}^*$ then the internal activity of each superplace in Y^* is independent of other activity in Y^*. Therefore the internal activity of each superplace is enabled in the reordered sequence. Further, since no external or terminal transition, occurs in $Y_1{}^*$ and since the outputs of a superplace are not decreased by the internal activity of the superplace, then the internal activity of each supertransition is only dependent on the activity of superplaces input to the supertransition. Now we know X_1 is ordered so that each node representing a superplace appears before the nodes representing supertransitions that are input to the superplace. Therefore $Y_1{}^*$ is ordered so that the internal activity of each superplace appears before the internal activity of its input supertransitions. Therefore the internal activity of each supertransition is enabled in the reordered sequence. Therefore the reordering does not affect the enabling, that is $M[Y_1{}^*\rangle M_1$.

($\Leftarrow$) Now suppose $\exists\, Y^* \in orderedInternalSeq(X_1) \; : \; M[Y^*\rangle M_1$.
$\Rightarrow Y^* = Y_{x_1''}{}^* Y_{x_2''}{}^* \cdots Y_{x_n''}{}^*$ such that $\forall i \in \{1,\ldots,n\} \; : \; Y_{x_i''}{}^* \in \sigma\mathbb{Y}_{x_i''}$ by Definition 7.19.
$\Rightarrow Y^* \in \sigma\mathbb{Y}_{X''}$.
$\Rightarrow M_1 \in$ *internallyReachable*(M,X). $\Diamond$

Having discussed how the unfolded RNSS can be constructed from the RNSS, we now prove that the unfolded graph is isomorphic to the full reachability graph. It is useful to first prove Lemmas 7.21 and 7.22. Lemma 7.21 tells us that any marking represented by a global vertex is a reachable marking of the full reachability graph. Lemma 7.22 says that a marking is internally reachable from a marking corresponding to a global vertex if and only if it is reachable in the full reachability graph.

Lemma 7.21. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, the RNSS of N, $(\mathcal{G}, \mathcal{G}_{X''})$, a global vertex $v \in \mathcal{V}_{\mathcal{G}}$, and a marking $M \in \mathbb{M}$ then:

$$M \in v^{-\natural} \Rightarrow M \in \mathbb{M}_R$$

Proof.
Since vertex $v \in \mathcal{V}_{\mathcal{G}}$ then Definition 7.16 implies that $v \in [[M_0\rangle\rangle^{\natural} \cup \{M_0^{\natural}\}$. Therefore there exists a marking $M_1 \in [[M_0\rangle\rangle \cup \{M_0\}$ such that $M_1{}^{\natural} = v$. Since $M_1 \in [[M_0\rangle\rangle \cup \{M_0\}$ then Definition 7.14 implies $M_1 \in \mathbb{M}_R$.

Now since $M \in v^{-\natural}$ then $M^{\natural} = v = M_1{}^{\natural}$. Thus for all supernodes $x'' \in X''$ the supernode markings $M|_{x''}$ and $M_1|_{x''}$ are in the same SCC in the supernode graph of x'' (i.e. $\forall x'' \in X'' : M|_{x''}{}^{c} = M_1|_{x''}{}^{c}$), and the marking of external places of M and M_1 are the same (i.e. $M|_{EP} = M_1|_{EP}$). Therefore M is reachable from M_1. Since M_1 is reachable and M is reachable from M_1 it follows that M is reachable. ◊

Lemma 7.22. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, and the RNSS of N, $(\mathcal{G}, \mathcal{G}_{X''})$, then:

$$M \in \mathbb{M}_R \Leftrightarrow \exists\, v \in \mathcal{V}_{\mathcal{G}} : M \in [[v^{-\natural}\rangle$$

Proof.
($\Rightarrow$) $M \in \mathbb{M}_R$ implies there exists a step sequence $Y^* \in \sigma\mathbb{Y}$, such that $M_0[Y^*\rangle M$. We show using induction that we can reorder the sequence Y^* so that every marking M_i following the occurrence of an external or terminal transition, corresponds to a global vertex. The required result immediately follows.

Inductive Proposition: A sequence Y^* as defined above, such that the number of firing elements in Y^* involving an external or terminal transition is n, can be reordered to a sequence $Y_1{}^* = (t_1, c_1) \ldots (t_m, c_m)$ such that $M_0[(t_1, c_1)\rangle M_1 \ldots [(t_m, c_m)\rangle M_m = M$ and for $0 < i \leq m$ if $(t_i, c_i) \in FE|_{ETT}$ then M_i corresponds to a global vertex (i.e. $M_i{}^{\natural} \in \mathcal{V}_{\mathcal{G}}$).

Basis: First consider the case where $n = 0$.
In this case there is no occurrence of an external or terminal transition in Y^*, and so M is internally reachable from the global vertex $M_0{}^{\natural}$.

Inductive Assumption: Assume that when $n = k$, $k > 0$, then Y^* can be reordered so that every marking following the occurrence of a firing element corresponds to a global vertex.

Inductive Step: Consider the case where $n = k + 1$. We know from the inductive assumption that the sequence can be reordered so that the marking, M_k, following the k-th firing element from $Y_1{}^*$ such that the firing element is in $FE|_{ETT}$, corresponds to a global vertex. Consider the remainder of the sequence $Y_r{}^* = M_k[(t_{k+1}, c_{k+1})\rangle \ldots [(t_m, c_m)\rangle M$. This sequence contains exactly one firing element from $FE|_{ETT}$, say (t_x, c_x). There are two cases: either t_x is an external transition or t_x is a terminal transition.

i. Suppose t_x is an external transition, that is $t_x \in ET$. We know that the internal activity of superplaces input to t_x is independent of internal activity of other supernodes. Therefore the sequence $Y_r{}^*$ can be reordered so that the internal activity of superplaces input to t_x occurs first, then (t_x, c_x) occurs, and the internal activity of supernodes not input to t_x occurs afterwards. Since Definition 7.16 adds a vertex to the global graph following internal activity of superplaces input to a transition followed by the occurrence of the transition, then the occurrence of (t_x, c_x) in this reordered sequence leads to a marking corresponding to a global vertex in $\mathcal{V}_\mathcal{G}$.

ii. Suppose t_x is a terminal transition of some supertransition, $t'' \in T''$. That is $t_x \in terminal(N_{t''})$. We know that the internal activity of the superplaces input to the supertransition of t'' is independent of the internal activity of other supernodes. We also know that the internal activity of the supertransition of t'' is only dependent on the internal activity of superplaces input to it. Therefore the sequence $Y_r{}^*$ can be reordered so that the internal activity of superplaces input to the supertransition of t'' occurs first, then internal activity of the supertransition of t'' occurs, then the terminal firing element (t_x, c_x) occurs, and finally internal activity of supernodes not input to the supertransition of t'' occurs. Since Definition 7.16 adds a vertex to the global graph following a sequence of of steps involving transitions internal to superplaces input to the supertransition of t'' and transitions internal to the supertransition of t'', followed by (t_x, c_x), then the occurrence of (t_x, c_x) in this reordered sequence leads to a marking corresponding to a vertex in $\mathcal{V}_\mathcal{G}$.

In both cases we can reorder the remainder of the sequence so that the marking following the firing element from $FE|_{ETT}$ corresponds to a global marking. Therefore we can reorder the sequence Y^* as required, and the inductive proposition holds for $n = k + 1$.

Conclusion: The inductive proposition holds for $n = 0$ and if we assume $n = k$ then it holds for $n = k + 1$ therefore by the principle of mathematical induction the proposition holds.

The above proposition tells us we can reorder Y^* so that each marking following an external or terminal transition corresponds to a global vertex, and M is internally reachable from the global vertex corresponding to the last external or terminal transition in the reordered sequence.

$(\Leftarrow)$
$\exists\, v \in \mathcal{V}_G : M \in [[v^{-t}\rangle$
$\Rightarrow \exists\, v \in \mathcal{V}_G : \exists\, M_1 \in \mathbb{M} : M_1{}^t = v \wedge M \in [[M_1\rangle$ by Definition 7.11

Since $M_1{}^{\phi} = v$ then by Definition 7.8 $M_1 \in v^{-\phi}$. Thus by Lemma 7.21 M_1 is reachable. It follows that since M is internally reachable from M_1 then M is reachable. ◊

We now prove that the unfolded RNSS is isomorphic to the full reachability graph.

Theorem 7.23. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, and the RNSS of N, then the unfolded graph of the RNSS, $(\mathcal{V}_u, \mathcal{E}_u)$, is isomorphic to the full reachability graph, $(\mathcal{V}, \mathcal{E})$.

Proof.
Since $\mathcal{V}_u = \bigcup_{v \in \mathcal{V}_G} [[v^{-\phi}\rangle$ then $M \in \mathcal{V} \Leftrightarrow M \in \mathcal{V}_u$ follows directly from Lemma 7.22.
We now prove $(M_1,(t,c),M_2) \in \mathcal{E} \Leftrightarrow (M_1,(t,c),M_2) \in \mathcal{E}_u$
There are two cases: (i) t is an external or terminal transition, or (ii) t is an internal transition:

i. Suppose t is an external or terminal transition, that is: $t \in ETT$.
($\Rightarrow$) Since M_1 and M_2 are reachable then by Lemma 7.22 there exists $v_1, v_2 \in \mathcal{V}_G$ such that $M_1 \in [[v_1^{-\phi}\rangle$ and $M_2 \in [[v_2^{-\phi}\rangle$. Internal activity of supernodes that do not affect the enabling of t may be required to reach M_1 from $v_1^{-\phi}$. However since (t,c) is enabled at M_1 then there is a marking, say M_1', reachable from $v_1^{-\phi}$ by transitions local only to supernodes that affect the enabling of t such that (t,c) is enabled at M_1', and the marking of the supernodes adjacent to t in M_1' is the same as the marking of the supernodes adjacent to t in M_1. Further the occurrence of (t,c) at M_1' leads to some marking M_2', and the marking of the supernodes adjacent to t in M_2' is the same as the marking of the supernodes adjacent to t in M_2. In other words, there exists $M_1' \in [_t[v_1{}^{-\phi}\rangle$ such that $M_1'[(t,c)\rangle M_2'$ for some $M_2' \in \mathbb{M}$ where $M_1'|_{^\circ t^\circ} = M_1|_{^\circ t^\circ}$ and $M_2'|_{^\circ t^\circ} = M_2|_{^\circ t^\circ}$. Definition 7.16 adds an edge to the global graph for the occurrence of an external or terminal transition from a marking reachable from a global vertex by transitions local to supernodes that affect the enabling of t. Therefore there exists $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_G$ such that $M_1'|_{^\circ t^\circ} = m_1$ and $M_2'|_{^\circ t^\circ} = m_2$. Therefore there exists $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_G$ such that $M_1 \in [[v_1^{-\phi}\rangle$ and $M_2 \in [[v_2^{-\phi}\rangle$ and $M_1|_{^\circ t^\circ} = m_1$, and $M_2|_{^\circ t^\circ} = m_2$.
Now since $M_1[(t,c)\rangle M_2$ then the marking M_1 of places not adjacent to t is equal to the marking M_2 of places not adjacent to t (since the occurrence of t only affects places adjacent to it). Therefore $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$.

Hence we have $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_G$ such that $M_1 \in [[v_1^{-\phi}\rangle$ and $M_2 \in [[v_2^{-\phi}\rangle$ and $M_1|_{^\circ t^\circ} = m_1$, and $M_2|_{^\circ t^\circ} = m_2$ and $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$. Therefore $(M_1,(t,c),M_2) \in \mathcal{E}_1$ of Definition 7.17. Therefore $(M_1,(t,c),M_2) \in \mathcal{E}_u$.

($\Leftarrow$) Since $t \in ETT$ then $(M_1,(t,c),M_2) \in \mathcal{E}_1$ of Definition 7.17.
$\Rightarrow \exists\, (v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_G$ of Definition 7.16, where $M_1 \in [[v_1^{-\phi}\rangle$ and $M_2 \in [[v_2^{-\phi}\rangle$ and $M_1|_{^\circ t^\circ} = m_1$ and $M_2|_{^\circ t^\circ} = m_2$, and $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$.
Since $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_G$ then by Definition 7.16 there is a marking, say M_1', reachable from $v_1^{-\phi}$ by transitions local only to supernodes that affect the enabling of t such that (t,c) is enabled at M_1', and the marking of the supernodes adjacent to t in M_1' is equal to m_1. Further the occurrence of (t,c) at M_1' leads to some

marking $M_2' \in v_2^{-t}$, and the marking of the supernodes adjacent to t in M_2' is equal to m_2. In other words, by Definition 7.16 there exists $M_1' \in [_t[v_1^{-t}\rangle$ and $M_2' \in v_2^{-t}$ where $m_1 = M_1'|_{^\circ t^\circ}$ and $m_2 = M_2'|_{^\circ t^\circ}$ and $M_1'[(t,c)\rangle M_2'$. Since $M_1'[(t,c)\rangle M_2'$ and $M_1'|_{^\circ t^\circ} = m_1 = M_1|_{^\circ t^\circ}$ and $M_2'|_{^\circ t^\circ} = m_2 = M_2|_{^\circ t^\circ}$, and $M_1|_{(P-^\circ t^\circ)} = M_2|_{(P-^\circ t^\circ)}$ then $M_1[(t,c)\rangle M_2$. Further, since M_1 is internally reachable from the global vertex v_1 respectively, then Lemma 7.22 implies M_1 is reachable.
Therefore $(M_1,(t,c),M_2) \in \mathcal{E}$.

ii. Suppose t is internal to a supernode of $x'' \in X''$, and is not a terminal transition, that is: $t \in T_{x''} - terminal(N_{x''})$.
($\Rightarrow$) Since$(M_1,(t,c),M_2) \in \mathcal{E}$ then $M_1[(t,c)\rangle M_2$.
$\Rightarrow M_1|_{x''}[(t,c)\rangle M_2|_{x''}$
Now by Lemma 7.22 there exists $v_1 \in \mathcal{V}_G$ such that $M_1 \in [[v_1^{-t}\rangle$
Since $M_1 \in [[v_1^{-t}\rangle$ then by Definition 7.16 $M_1|_{x''} \in \mathcal{V}_{x''}$.
Since $M_1|_{x''}[(t,c)\rangle M_2|_{x''}$ and $M_1|_{x''} \in \mathcal{V}_{x''}$ then Definition 7.16 implies
$(M_1|_{x''},(t,c),M_2|_{x''}) \in \mathcal{E}_{x''}$.
$\Rightarrow (M_1,(t,c),M_2) \in \mathcal{E}_2$ of Definition 7.17.
$\Rightarrow (M_1,(t,c),M_2) \in \mathcal{E}_u$ by Definition 7.17.

($\Leftarrow$) Since $t \in T - ETT$ then $(M_1,(t,c),M_2) \in \mathcal{E}_2$ of Definition 7.17.
$\Rightarrow \exists\, (m_1,(t,c),m_2) \in \mathcal{E}_{x''}$ of Definition 7.16 for some $x'' \in X''$ such that $M_1|_{x''} = m_1$, and $M_2|_{x''} = m_2$ and $M_2 = M_1 - m_1 + m_2$, where M_1 and M_2 are internally reachable from a global vertex. Since M_1 and M_2 are internally reachable from a global vertex then Lemma 7.22 implies M_1 and M_2 are reachable. Since $M_2 = M_1 - m_1 + m_2$ and $m_1[(t,c)\rangle m_2$ then $M_1[(t,c)\rangle M_2$. Therefore $(M_1,(t,c),M_2) \in \mathcal{E}$.

$\Diamond$

7.4.8 The RNSS Algorithm

In this section we present an algorithm that constructs the RNSS (Algorithm 7.5). We refer to this algorithm as the *RNSS algorithm*. For clarity, the RNSS algorithm presented here does not make use of the abstract graph to limit the number of refined firing elements considered, and therefore does not require the abstract graph. However, since node refinement is a system morphism then, as was the case for the algorithms that cater for type and subnet refinement, the RNSS algorithm can be modified to make use of the abstract graph to limit the number of refined firing elements considered. The algorithm that caters for all three forms of refinement (Algorithm 7.9) presented later in this chapter does just that.

The structure of the RNSS algorithm (Algorithm 7.5) is the same as that of the standard reachability graph algorithm (Algorithm 7.1), but since the global graph stores global vertices then: the *Waiting* set stores global vertices; the function ADDVERTEX adds a global vertex to the global graph; the function MATCH$(\mathcal{G},v)$ returns true if and only if the vertex v is equal to any vertex in the global graph; and the function ADDEDGE adds an edge labelled with source and successor markings to the global graph.

Algorithm 7.5 The RNSS algorithm

```
RNSS(G, G_X'', N, M, possible)
begin
  Waiting := ∅
  v := GLOBALVERTEX(G_X'', M)
  if not MATCH(G, v) then
    ADDVERTEX(G, v)
    Waiting := {v}
  end if
  while Waiting ≠ ∅ do
    v1 := SELECT(Waiting)
    for all (v1, (m1, (t, c), m2), v2) ∈ EDGESFROM-NODE(N, N', G_X'', v1, possible) do
      if not MATCH(G, v2) then
        ADDVERTEX(G, v2)
        Waiting := Waiting + {v2}
      end if
      ADDEDGE(G, (v1, (m1, (t, c), m2), v2))
    end for
    Waiting := Waiting − {v1}
  end while
  return (G, G_X'')
end
```

An algorithm for the GLOBALVERTEX function is presented in Algorithm 7.6. It takes the set of supernode graphs $\mathcal{G}_{X''} = \{G_{x'} \mid x' \in X''\}$, and the marking M, and calculates the global vertex representing M. That is, it calculates M^t. The COMPUTESCCS$(\mathcal{G}_{x'}, M|_{x'})$ function computes the SCCs of the vertices reachable from $M|_{x'}$ in the supernode graph $\mathcal{G}_{x'}$, and returns the SCC index of the marking $M|_{x'}$. Thus the global vertex, v, representing M is calculated by firstly initialising it to $M|_{EP}$ (where EP is the set of external places). The state space of each supernode, x', is then developed from $M|_{x'}$, and the pair of x' and SCC of $M|_{x'}$ is added to v. In other words v is formed by replacing the marking of each supernode by its SCC index.

Since each supernode has an associated graph, and since the local state space is developed from different starting points, then it follows that the supernode graph is not necessarily connected. Also, it may be the case that part or all of local state space of a given supernode has previously been developed. This is clearly an advantage, saving time and space as the state space does not need to be developed again (as would be the case in the standard algorithm). We also note that the supernode graphs are independent, and the implementation could therefore develop them in parallel.

The function EDGESFROM-NODE$(N, N', \mathcal{G}_{X''}, v, possible)$, presented in Algorithm 7.7, returns the global edges and global successors from the global vertex v for any external or terminal transition in the set $possible \subseteq ETT$. For this function to return all the global edges from v, the set *possible* must include all external and terminal transitions that are enabled at a marking internally reachable from v. We do not indicate how this set is calculated. All functions of Algorithm 7.7 have been previously presented: the functions FINDANINVERSE and INTERNALLYREACHABLE are as presented in Algorithm 7.4; the func-

tions TERMINAL and INPUTSUPERPLACES are as described in Definitions 7.2 and 7.12 respectively; and the functions REACHABILITYGRAPH, and ENABLEDFIRINGELEMENTS are as presented for the standard reachability graph algorithm (Section 7.1).

The algorithm for the EDGESFROM-NODE$(N, N', \mathcal{G}_{X''}, v, possible)$ function first uses the FINDANINVERSE function (described in Algorithm 7.4) to construct a marking M such that $M^t = v$. It then considers all external transitions and then all supertransitions. For each external transition, t, it adds an edge to the global graph for the occurrence of t from a marking internally reachable in superplaces input to t from M. Since the local graphs of superplaces are developed when the global vertex is added (see the function GLOBALVERTEX) then the INTERNALLYREACHABLE function (of Algorithm 7.4) can be used to find the internally reachable markings in the superplaces input to t.

For each supertransition the EDGESFROM function first considers the markings internally reachable in superplaces input to the supertransition. For each such marking, M_1, it develops the local reachability graph of the supertransition from the marking M_1 restricted to the places of the supertransition (including environment places). Since the internally reachable markings of the supertransition are considered from each marking internally reachable in the superplaces input to the supertransition then it follows from Proposition 7.20 that the supertransition graph will contain the supertransition component of all markings internally reachable from M. The EDGESFROM function then adds an edge to the global graph for the occurrence of a terminal transition of the supertransition from a marking internally reachable in the supertransition from M_1.

Algorithm 7.6 The GLOBALVERTEX function

GLOBALVERTEX$(\mathcal{G}_{X''}, M)$
begin
  $v := M|_{EP}$
  for all $x'' \in X''$ **do**
    REACHABILITYGRAPH$(\mathcal{G}_{x''}, N_{x''}, M|_{x''})$
    $scc :=$ COMPUTESCCS$(\mathcal{G}_{x''}, M|_{x''})$
    $v := v + (x'', scc)$
  end for
  return M^t
end

Algorithm 7.7 The EDGESFROM-NODE function

```
EDGESFROM-NODE(N, N', G_X'', v, possible)
begin
  Result := ∅
  M := FINDANINVERSE(v)
  for all t ∈ (ET ∩ possible) do
    for all M_1 ∈ INTERNALLYREACHABLE(M, INPUTSUPERPLACES(t)) do
      for all (t,c) ∈ ENABLEDFIRINGELEMENTS(N, M_1, t) do
        M_2 := M_1 − E^−((t,c)) + E^+((t,c))
        v_2 := GLOBALVERTEX(M_2)
        Result := Result + {(v, (M_1|°t°, (t,c), M_2|°t°), v_2)}
      end for
    end for
  end for
  for all t'' ∈ T'' do
    for all M_1 ∈ INTERNALLYREACHABLE(M, INPUTSUPERPLACES(t'')) do
      REACHABILITYGRAPH(G_t'', N_t'', M_1|t'')
      COMPUTESCCS(G_t'', N_t'', M_1|t'')
      for all M_2 ∈ INTERNALLYREACHABLE(M_1, {t''}) do
        for all t ∈ (TERMINAL(N_t'') ∩ possible) do
          for all (t,c) ∈ ENABLEDFIRINGELEMENTS(N, M_2, t) do
            M_3 := M_2 − E^−((t,c)) + E^+((t,c))
            v_2 := GLOBALVERTEX(M_3)
            Result := Result + {(v, (M_2|°t°, (t,c), M_3|°t°), v_2)}
          end for
        end for
      end for
    end for
  end for
  return Result
end
```

7.4.9 Properties of the RNSS

We now define how the various dynamic properties (reachability, dead markings, home properties, liveness, and boundedness) can be determined directly from the RNSS. This is particularly important since unfolding the RNSS may be an expensive operation in terms of time and/or space. The propositions of this section are based on those of Christensen and Petrucci [51]. The RNSS is compared to the Modular State Space of Christensen and Petrucci in Section 7.4.16. We use the net of Figure 7.3 and its associated state space (Figure 7.4) as an example throughout this section.

In the following we use the functions *Term*, *Trivial*, and *FiringElements*. The function *Term* takes a set of SCCs and returns the SCCs that are terminal (i.e. those that have no outgoing arcs) and the function *Trivial* which takes a set of SCCs and returns the set of trivial SCCs (i.e. those that have exactly one vertex and no edges). The function *FiringElements* maps a SCC into the set of firing elements that occur in the labels of the edges of the component. Similarly, we use *FiringElements* to map a set of reachable markings to the set of firing elements that occur in the labels of the edges between two vertices of the set. We use SCC_G to denote the set of strongly connected components of the global graph. Similarly, $SCC_{x''}$ denotes the set of SCCs of the supernode graph of $x'' \in X''$.

The propositions concerning the dynamic properties commonly need to know if a given marking is internally reachable from a marking in the set v^{-t} where v is a global vertex. As we discussed in Section 7.4.7, since for each supernode, the supernode marking of all markings in v^{-t} are in the same SCC then the set of markings internally reachable from markings in v^{-t} is equal to the markings internally reachable from any M_v, where $M_v{}^t = v$. We can construct such a marking M_v using the FINDANINVERSE function of Algorithm 7.4. Having constructed M_v, we can then use the algorithm for the INTERNALLYREACHABLE function (presented in Algorithm 7.4) to find the set of markings internally reachable from it. However, if we just want to know if a marking is internally reachable from M_v then we do not have to find this set. Proposition 7.25 (a) allows us to optimise the algorithm for the INTERNALLYREACHABLE function so that we can efficiently determine if a marking is internally reachable from M_v.

Similarly the propositions for dead markings, home properties, and liveness require the set of markings internally reachable from a global vertex such that for each marking M in the set, the marking M restricted to each supernode is in a terminal SCC of the supernode graph. Again we do not have to construct all markings internally reachable from a marking representing a global vertex to find this set. Proposition 7.25 (b) allows us to optimise the algorithm for the INTERNALLYREACHABLE function so we can efficiently find the internally reachable markings where each supernode component is in a terminal SCC of the supernode graph.

Definition 7.24. Given a supernode $x'' \in X''$ we define the function $sep() : X'' \rightarrow 2^P$ to return the places internal to the supernode of x'' that are not input to a supertransition. That is, *sep* returns the places of the supernode of x'' that are *separate* from any supertransition:

$$sep(x'') = \{p \in P_{x''} \mid \forall t'' \in T'' \ : \ p \notin inpenv(N_{t''})\}$$

We define:

$$nonsep(x'') = P_{x''} - sep(x'')$$

Note: $sep(x'')$ returns the places of the supernode of x'' that are *separate* from all supertransitions. That is, $sep(x'')$ returns the set of places that are not input to a supertransition. $nonsep(x'')$ returns the set of places of the supernode of x'' that are input to a supertransition. Clearly if x'' represents a supertransition (i.e $x'' \in T''$) then $sep(x'')$ is the set of all places of the supertransition.

Proposition 7.25. For a net N with RNSS = $(\mathcal{G}, \mathcal{G}_{X''})$, a set of nodes $X \subseteq X''$, an ordered permutation of X, $X_1 = order(X) = x_1'' \cdots x_n''$, and a marking $M_v \in \mathbb{M}$ then:

a. $M \in internallyReachable(M_v, X) \Leftrightarrow$
$\exists Y_{x_1''}^* \cdots Y_{x_n''}^* \in orderedInternalSeq(X_1) \ : \ M_v[Y_{x_1''}^*\rangle M_1 \cdots [Y_{x_n''}^*\rangle M_n = M$
$\wedge \ (\forall i \in \{1,\ldots,n\} \ : \ (M_i|_{sep(x_i'')} = M|_{sep(x_i'')}) \wedge (M_i|_{nonsep(x_i'')} \geq M|_{nonsep(x_i'')}))$

b. $M \in internallyReachable(M_v, X) \wedge (\forall i \in \{1,\ldots,n\} \ : \ ((M|_{x_i''})^c \in Term(SCC_{x_i''})) \Leftrightarrow \exists Y_{x_1''}^* \cdots Y_{x_n''}^* \in orderedInternalSeq(X_1) \ :$
$M_v[Y_{x_1''}^*\rangle M_1 \cdots [Y_{x_n''}^*\rangle M_n = M$
$\wedge \ (\forall i \in \{1,\ldots,n\} \ : \ ((M_i|_{x_i''})^c \in Term(SCC_{x_i''})))$

Note:

a. M is internally reachable from M_v in the supernodes of X if and only if there is an ordered internal sequence (see Definition 7.19) such that each marking reached following internal activity of a supernode, M_i, restricted to the separated places of the supernode is equal to M restricted to the separated places of the supernode, and M_i restricted to the non-separated places of the supernode is greater than or equal to M restricted to the non-separated places of the supernode.

b. M is internally reachable from M_v and each supernode marking of M is in a terminal SCC if and only if there is an ordered internal sequence (see Definition 7.19) such that each marking reached following internal activity of a supernode, restricted to the marking of the supernode is in a terminal SCC of the supernode graph.

Proof.

a. ($\Rightarrow$) $M \in internallyReachable(M_v, X)$
$\Rightarrow \exists Y^* = Y_{x_1''}^* \cdots Y_{x_n''}^* \in orderedInternalSeq(X_1) \ : \ M_v[Y_{x_1''}^*\rangle M_1 \cdots [Y_{x_n''}^*\rangle M_n = M$
by Proposition 7.20. Since Y^* does not contain the occurrence of any external or terminal transitions then the marking of separated places of the supernode of x_i'' are only affected by internal activity of x_i''. Further the marking of each non-separated

place is only affected by the occurrence of input border transitions of supertransitions, which can only remove tokens from the non-separated places. Therefore since Y^* leads from M_v to M we have $\forall i \in \{1,\ldots,n\} : (M_i|_{sep(x_i'')} = M|_{sep(x_i'')}) \wedge (M_i|_{nonsep(x_i'')} \geq M|_{nonsep(x_i'')})$.

($\Leftarrow$) Since $Y_{x_1''}{}^* \cdots Y_{x_n''}{}^*$ is an ordered internal sequence then the required result follows directly from Proposition 7.20.

b. $M \in internallyReachable(M_v, X) \Leftrightarrow$
$\exists Y_{x_1''}{}^* \cdots Y_{x_n''}{}^* \in orderedInternalSeq(X_1) : M_v[Y_{x_1''}{}^*\rangle M_1 \cdots [Y_{x_n''}{}^*\rangle M_n = M$ by Proposition 7.20. So we must show:
$\forall i \in \{1,\ldots,n\} : (M|_{x_i''})^c \in Term(SCC_{x_i''}) \Leftrightarrow (M_i|_{x_i''})^c \in Term(SCC_{x_i''})$.
There are two cases:

i. $x_i'' \in T''$.
Since the separated places of x_i'' are all places internal to the supertransition of x_i'' (i.e. $sep(x_i'') = P_{x_i''}$), and the internal activity of superplaces input to the supertransition of x_i'' occurs in Y^* before the internal activity of the supertransition of x_i'', then $M_i|_{x_i''} = M|_{x_i''}$. Therefore $(M|_{x_i''})^c \in Term(SCC_{x_i''})) \Leftrightarrow (M_i|_{x_i''})^c \in Term(SCC_{x_i''})$.

ii. $x_i'' \in P''$.
Since a superplace cannot remove tokens from its output border places, then the markings of the non-separated places will be constant over all the markings in a given SCC of the superplace of x_i''. Further, since $(M_i|_{sep(x_i'')} = M|_{sep(x_i'')})$ then only the marking of non-separated places is different between $M_i|_{x_i''}$ and $M|_{x_i''}$. Therefore $(M|_{x_i''})^c \in Term(SCC_{x_i''})) \Leftrightarrow (M_i|_{x_i''})^c \in Term(SCC_{x_i''})$.

$\Diamond$

It follows from Proposition 7.25 (a) that if we want to determine if a marking M is internally reachable from a given marking, M_v, in a given set of supernodes, $X \subseteq X''$, then we only have to consider those ordered internal sequences $Y_{x_1''}{}^* \cdots Y_{x_n''}{}^*$ from the set $orderedInternalSeq(order(X))$ such that $M_v[Y_{x_1''}{}^*\rangle M_1 \cdots [Y_{x_n''}{}^*\rangle M_n = M$ and such that for each marking M_i ($0 < i \leq n$) restricted to the separated places of the supernode is equal to M restricted to the separated places of the supernode, and M_i restricted to the non-separated places of the supernode is greater than or equal to M restricted to the non-separated places of the supernode.

Recall that the INTERNALLYREACHABLE(M_v, X) function of Algorithm 7.4 returns the set of markings internally reachable from M_v by transitions local the supernodes of X. It follows from Proposition 7.25 (a) that if we only want to know if a marking M is internally reachable from M_v then we can optimise the INTERNALLYREACHABLE algorithm. The optimisation is to change the LOCALLYREACHABLE($\mathcal{M}, x_i''$) function so that it returns the set of markings locally reachable in the supernode of x_i'' from a marking in $\mathcal{M}$ where each marking M_j in the set returned is such that $M_j|_{sep(x_i'')} = M|_{sep(x_i'')}$ and $M_j|_{nonsep(x_i'')} \geq M|_{nonsep(x_i'')}$. The marking M is then internally reachable from M_v if it is an element of the result set returned by the optimised algorithm. Clearly this optimisation can save considerable effort since the set of markings for which locally reachable markings have to be found is reduced to only those markings that can potentially lead to M. In the sections on determining dynamic properties without unfolding that follow whenever

we need to determine whether a marking is internally reachable from a global vertex we assume this optimised algorithm is used.

Similarly, Proposition 7.25 (b) allows us to optimise the algorithm for the INTERNALLYREACHABLE(M_v, X) function (Algorithm 7.4) to find the markings that are internally reachable from a given marking and are such that the marking restricted to each supernode is in a terminal SCC of the supernode graph. In this case we modify the LOCALLYREACHABLE($\mathcal{M}, x_i''$) function so that it returns the set of markings locally reachable from a marking in $\mathcal{M}$ where each marking M_j in the set returned by LOCALLYREACHABLE is such that $M_j|_{x_i''}$ is in a terminal SCC of the supernode graph of x_i''. The optimised algorithm will return a set containing the markings that are internally reachable from M_v such that the marking restricted to any supernode of X is in a terminal SCC of the supernode graph. This optimisation reduces the markings for which locally reachable markings have to be found to only those markings that can lead to a marking where each supernode component is in a terminal SCC of the supernode graph. In the sections on determining dynamic properties without unfolding that follow whenever we need to determine the set of markings internally reachable from a global vertex such that the SCC of each supernode component of each marking is in a terminal SCC we assume this optimised algorithm is used.

We now consider how the dynamic properties can be determined from the RNSS without unfolding.

7.4.10 Reachability

For a full reachability graph, a marking $M \in \mathbb{M}$ is said to be *reachable* from the initial marking M_0 if there exists a finite directed path from M_0 to M. Given a RNSS of a net N we wish to determine whether a marking M is reachable without unfolding.

Proposition 7.26. For a net N with RNSS = $(\mathcal{G}, \mathcal{G}_{X''})$, $M \in \mathbb{M}$ then:

a. $M \in \mathbb{M}_R \Leftrightarrow \exists v \in \mathcal{V}_{\mathcal{G}} : M \in [[v^{-t}\rangle$

b. $M \in \mathbb{M}_R \Rightarrow \forall x'' \in X'' : M|_{x''} \in \mathcal{V}_{x''}$

Explanation:

a. A marking is reachable iff it is internally reachable from a global vertex.

b. If a marking M is reachable then for each supernode of $x'' \in X''$, the restriction of M to x'' is in the vertices of the supernode graph of x''.

Proof.

a. Holds by Lemma 7.22.

b. From part (a) we know $M \in \mathbb{M}_R \Leftrightarrow \exists v \in \mathcal{V}_{\mathcal{G}} : M \in [[v^{-t}\rangle$.
Since Definition 7.16 adds a vertex to the supernode graph of x'' for every marking internally reachable from a global vertex, then $M \in [[v^{-t}\rangle$ implies $\forall x'' \in X'' : M|_{x''} \in \mathcal{V}_{x''}$. Therefore $M \in \mathbb{M}_R \Rightarrow \forall x'' \in X'' : M|_{x''} \in \mathcal{V}_{x''}$.

$\Diamond$

Sketch of algorithm: Proposition 7.26 allows us to check whether a marking $M \in \mathbb{M}$ is reachable without unfolding. By Proposition 7.26(b) if any supernode component of M is not in the corresponding supernode graph (that is if $\exists\, x'' \in X'' : M|_{x''} \notin \mathcal{V}_{x''}$) then M is not reachable. Otherwise we check if M is internally reachable from a global vertex. By Proposition 7.26(a) if M is internally reachable from a global vertex then it is reachable, otherwise it is not.

Example: As an example, we apply the algorithm to the RNSS of Figure 7.4 (c) to determine if the marking $M = (p_2\text{-}out_1, x) + (p_2\text{-}p_2, x) + (p_4, y)$ is reachable. The state $(p_2\text{-}out_1, x) + (p_2\text{-}p_2, x)$ is in the supernode graph of p_2. Similarly the state (p_4, y) is in the supernode graph of t_3. So our first requirement is satisfied. We therefore check each global vertex to determine if M is internally reachable from it. Here we find that M is internally reachable from a marking represented by the global vertex v_1. Therefore M is reachable.

7.4.11 Dead Markings

A marking is *dead* if no transition is enabled for that marking. That is, marking M is dead if $\forall (t,c) \in FE : \neg M[(t,c)\rangle$. A set of firing elements, $X \subseteq FE$, is dead iff no element of the set can become enabled, that is iff $\forall M \in [M_0\rangle\ \forall (t,c) \in X : \neg M[(t,c)\rangle$

We now consider finding dead markings in a RNSS = $(\mathcal{G}, \mathcal{G}_{X''})$ without unfolding.

Proposition 7.27.

a. $M \in \mathbb{M}$ is dead $\Leftrightarrow$
$\Big(\forall x'' \in X'' : \forall (m_1, (t,c), m_2) \in \mathcal{E}_{x''} : m_1 \neq M|_{x''}\Big) \ \wedge$
$\Big(\forall (v_1, (m_1, (t,c), m_2), v_2) \in \mathcal{E}_{\mathcal{G}} : M|_{t^\circ} = m_1 \Rightarrow M \notin [[v_1^{-t}\rangle\Big)$

b. $M \in \mathbb{M}$ is dead $\Leftrightarrow$
$\Big(\forall x'' \in X'' : (M|_{x''})^c \in Term(SCC_{x''}) \cap Trivial(SCC_{x''})\Big) \ \wedge$
$\Big(\forall (v_1, (m_1, (t,c), m_2), v_2) \in \mathcal{E}_{\mathcal{G}} : M|_{t^\circ} = m_1 \Rightarrow M \notin [[v_1^{-t}\rangle\Big)$

c. no marking $M \in \mathbb{M}_R$ is dead $\Leftarrow$
$\exists\, x'' \in X'' : \forall\, m_1 \in \mathcal{V}_{x''} : \exists\, (m_1, (t,c), m_2) \in \mathcal{E}_{x''}$

d. no marking $M \in \mathbb{M}_R$ is dead $\Leftarrow$
$\exists\, x'' \in X'' : Term(SCC_{x''}) \cap Trivial(SCC_{x''}) = \emptyset$

Explanation:

a. A marking is dead iff there is no enabled internal transition $\Big(\forall x'' \in X'' : \forall (m_1, (t,c), m_2) \in \mathcal{E}_{x''} : m_1 \neq M|_{x''}\Big)$, and the marking does not enable an external or terminal transition.

b. A marking is dead iff it belongs to terminal and trivial components of all supernode graphs, and the marking does not enable an external or terminal transition.

c. We know that there is no reachable dead marking if there exists a supernode graph for which every vertex has an outgoing edge.

d. We know that there is no reachable dead marking if there exists a supernode graph without any SCCs being both terminal and trivial.

Proof.

a. Let M be a reachable marking. There is an edge from M by (t,c) in the unfolded graph iff the edge is in the set $\mathcal{E}_1$ or $\mathcal{E}_2$ of Definition 7.17. That is, there is an edge in the unfolded graph iff
$\Big(\exists\, x'' \in X'' : \exists\, (m_1,(t,c),m_2) \in \mathcal{E}_{x''} : m_1 = M|_{x''}\Big) \;\vee$
$\Big(\exists\, (v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \;:\; M|_{\circ_t^\circ} = m_1 \wedge M \in [[v_1^{-\phi}\rangle)\Big)$.
Therefore there is no edge from M in the unfolded graph iff
$\Big(\forall x'' \in X'' : \forall (m_1,(t,c),m_2) \in \mathcal{E}_{x''} : m_1 \neq M|_{x''}\Big) \;\wedge$
$\Big(\forall (v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \;:\; M|_{\circ_t^\circ} = m_1 \Rightarrow M \notin [[v_1^{-\phi}\rangle\Big)$.
Since by Proposition 7.23 the unfolded graph is isomorphic to the full graph then $M \in \mathbb{M}$ is dead iff
$\Big(\forall x'' \in X'' : \forall (m_1,(t,c),m_2) \in \mathcal{E}_{x''} : m_1 \neq M|_{x''}\Big) \;\wedge$
$\Big(\forall (v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \;:\; M|_{\circ_t^\circ} = m_1 \Rightarrow M \notin [[v_1^{-\phi}\rangle\Big)$.

b. Clearly there are no enabled transitions at terminal and trivial SCCs. Therefore right hand sides of (a) and (b) are equivalent.

c. Let us suppose that there exists a supernode $x'' \in X''$ such that all vertices of the supernode graph of x'' have at least one outgoing edge. By Definition 7.16 the supernode graph of x'' contains the restriction to x'' of all markings internally reachable from a global vertex, and by Lemma 7.22 every marking is internally reachable from a global vertex. Therefore for every reachable marking a transition internal to the supernode of x'' is enabled. Thus there is no reachable dead marking.

d. Again since there are no enabled transitions at terminal and trivial SCCs then the right hand sides of (c) and (d) are equivalent.

◊

Sketch of algorithm: We can use Proposition 7.27 (b) to find all reachable dead markings. Lemma 7.22 tells us that every reachable marking is internally reachable from a marking corresponding to a global vertex. Therefore we will consider all reachable markings by considering those markings internally reachable from a global vertex.

For each global vertex v_1, we consider each marking, M, internally reachable from the global vertex where for all supernodes the supernode component of M is in a both terminal and trivial SCC. M is then dead if for each edge $(v_1,(m_1,(t,c),m_2),v_2)$ of the global graph such that $M|_{\circ_t^\circ} = m_1$ then M is not internally reachable from v_1.

Example: We can apply this algorithm to the example of Figure 7.4. We begin with the global vertex v_0. The only marking internally reachable from the global vertex v_0 such that all supernode markings are in both a terminal and trivial SCC is $M = (p_1,x) + (p_2,0) + (t_3,5)$. There is the edge $(v_0,((p_2,0),(t_1,x),(p_2,1)),v_1)$ in the global graph, and since $M|_{\circ_{t_1}^\circ} = (p_2,0)$ and M is internally reachable from v_1 then M is not dead. Therefore there are no dead markings internally reachable from the global vertex v_0. Similarly there are no dead markings internally reachable from the other global vertices. We can conclude that there are no reachable dead markings.

7.4.12 Home Properties

Informally, a home marking is a marking to which it is always possible to return, that is $M_H \in \mathbb{M}_R$ is a home marking iff $\forall M' \in [M_0\rangle : M_H \in [M'\rangle$. A home space is a set of markings such that it is always possible to return to a marking of the set, that is $X \subseteq \mathbb{M}_R$ is a home space iff $\forall M' \in [M_0\rangle : X \cap [M'\rangle \neq \emptyset$.

Here we want to check whether a given (set of) reachable marking(s) is a home marking (space) or not. We first express the home properties for a marking and then for a set of markings.

Proposition 7.28.

a. $M_H \in \mathbb{M}_R$ is a home marking $\Leftrightarrow$

$$\Big(\forall scc \in Term(SCC_G) : \exists\, v \in scc : M_H \in [[v^{-\ell}\rangle\Big) \;\wedge$$
$$\Big(\forall v \in \mathcal{V}_G : \forall M \in [[v^{-\ell}\rangle : (\forall x'' \in X'' : (M|_{x''})^c \in Term(SCC_{x''}))$$
$$\Rightarrow M_H \in [[M\rangle \;\vee$$
$$\exists\,(v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_G \;:\; \exists\, M_1 \in [_{t_1}[v^{-\ell}\rangle \;:\; M_1|^{\circ}_{t_1\circ} = m_1\Big)$$

b. $X \subseteq \mathbb{M}_R$ is a home space $\Leftrightarrow$

$$\Big(\forall scc \in Term(SCC_G) : \exists\, v \in scc : X \cap [[v^{-\ell}\rangle \neq \emptyset\Big)$$
$$\wedge\; \Big(\forall v \in \mathcal{V}_G : \forall M \in [[v^{-\ell}\rangle : (\forall x'' \in X'' : (M|_{x''})^c \in Term(SCC_{x''}))$$
$$\Rightarrow (X \cap [[M\rangle \neq \emptyset) \;\vee$$
$$\exists\,(v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_G \;:\; \exists\, M_1 \in [_{t_1}[v^{-\ell}\rangle \;:\; M_1|^{\circ}_{t_1\circ} = m_1\Big)$$

Explanation:

a. A marking M_H is a home marking if every terminal SCC of the global graph contains a global vertex from which M_H is internally reachable $\Big(\forall scc \in Term(SCC_G) : \exists v \in scc : M_H \in [[v^{-\ell}\rangle\Big)$, and from every marking M internally reachable from a global vertex such that for all $x'' \in X''$ the marking $M|_{x''}$ is in a terminal SCC of the supernode graph of x'', then M_H is reachable from M by internal transitions or it is possible to reach a marking where an external or terminal transition is enabled.

b. The home space proposition is similar to (a) — we check for non-empty intersection rather than membership.

Proof.
(a) is a special case of (b) where X contains only one marking. Thus we will only prove (b).
($\Rightarrow$) Let X be a home space. Here we show the first conjunct of the right hand side holds. Let *scc* be a terminal SCC of the global graph, and M be a marking such that M^{ℓ} is a vertex of *scc*. From a global vertex in a terminal SCC it is only possible to reach other global vertices in the SCC. Therefore the only markings that can be reached from M are those that can be reached internally from a global vertex in *scc*. That is, the only markings that can be reached are those in $[[v^{-\ell}\rangle$ for any $v \in scc$. Since X is a home space then from any marking it is possible to reach a marking in X. Therefore there exists a vertex $v \in scc$ such that $X \cap [[v^{-\ell}\rangle \neq \emptyset$.

Now we show the second conjunct of the right hand side holds. Consider a marking M internally reachable from a global vertex, such that for all $x'' \in X''$ the marking $M|_{x''}$ is in a terminal SCC of the supernode graph of x''. The markings that can be reached from M are those that can be internally reached (i.e. those markings in $[[M\rangle)$ together with those that can be reached by the occurrence of an external or terminal transition (possibly followed by further internal and external transitions). Since X is a home space then either $X \cap [[M\rangle \neq \emptyset$ or there is an external or terminal transition enabled at a marking internally reachable from M. Proposition 7.23 tells us that for the occurrence of an external or terminal transition $t_1 \in ETT$ from any marking M_1 then there is an edge $(v_1,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}}$ such that $M_1 \in [_{t_1}[v_1^{-t}\rangle$ and $M_1|_{^\circ t_1^\circ} = m_1$. So we have $X \cap [[M\rangle \neq \emptyset \ \vee \ \exists\, (v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} : \exists\, M_1 \in [_{t_1}[v^{-t}\rangle \ : \ M_1|_{^\circ t_1^\circ} = m_1$, as required.

($\Leftarrow$) By Lemma 7.22 every marking is internally reachable from a global vertex. Consider some marking M internally reachable from a global vertex $v_1 \in \mathcal{V}_{\mathcal{G}}$. From M it is possible to internally reach a marking M_1, where for each $x'' \in X''$ the marking $M_1|_{x''}$ is in a terminal SCC of the supernode graph of x''. By the second conjunct of the condition we are guaranteed that a marking in X is internally reachable from M, or that an external or terminal transition is enabled at a marking internally reachable from M_1. By the occurrence of external or terminal transition we can obtain another marking, M_2, corresponding to another global vertex, $v_2 \in \mathcal{V}_{\mathcal{G}}$. Since v_2 is reached by the occurrence of an external or terminal transition then $v_2 \neq v_1$. From the global vertex v_2 the process described to obtain v_2 can be repeated until a terminal SCC of the global graph is found. From the first conjunct, one of its successors is in X. $\Diamond$

Sketch of algorithm: To check whether a given set of markings, X, is a home space, we first mark the vertices that satisfy the first conjunct of the condition. That is, we mark the global vertices in terminal SCCs of the global graph from which a marking of X is internally reachable (recall that the set of markings internally reachable from a marking M includes M and so global vertices corresponding to a marking of X are also marked). If there exists a terminal SCC of the global graph without any marked vertices then the first conjunct of the condition is not satisfied and therefore X is not a home space. Otherwise, we have to check the second conjunct of the condition. For each vertex in the global graph we take the internally reached markings such that every supernode marking is in a terminal SCC of the corresponding supernode graph. For each such marking, M, we check that either a marking in X is internally reachable from M, or that it is possible to reach a marking where an external or terminal transition is enabled. If one marking does not satisfy this requirement, X is not a home space, otherwise it is.

Example: As an example, we consider whether the set $X = \{M_0 = (p_1,x) + (p_4,y)\}$ is a home space of the net of Figure 7.3. There are no global vertices in the terminal SCCs of the global graph from which M_0 is internally reachable, and so the first conjunct of the condition is not satisfied. Therefore X is not a home space. Further, since there is no single marking reachable from all vertices of the terminal SCCs of the global graph then the first conjunct of the condition will never be satisified for a single marking (i.e. for a home space of size 1). Thus the net has no home marking.

We now consider whether the set $X_1 = \{M_1 = (p_2 - p_1,x) + (p_3,x+y), M_2 = (p_2 - p_1,x) + (p_2 - out1,x) + (p_5,y)$ is a home space. Here M_1 is internally reachable from the global vertex v_3 and M_2 is internally reachable from the global vertex v_4, so we mark both v_3 and v_4. Now all terminal SCCs of the global graph are marked so we must check the

second conjunct of the condition. For each vertex of the global graph we consider the internally reached markings such that every supernode marking is in a terminal SCC of the corresponding sueprnode graph. For each of these markings either M_1 or M_2 can be reached or an external or terminal transition is enabled. Hence X_1 is a home space.

7.4.13 Liveness

Intuitively, liveness tells us that a (set of) firing element(s) remains active. That is, a set of firing elements, $X \subseteq FE$, is *live* if and only if from each reachable marking a marking can be reached where a firing element from X is enabled. That is, X is live if $\forall M \in \mathbb{M}_R \; \exists M' \in [M\rangle \; \exists (t,c) \in X \; : \; M'[(t,c)\rangle$ (Jensen [102, Def 4.10 (iii)]).

Given a RNSS, we now show how to determine whether a given set of transitions is live without unfolding. We first express the liveness properties for a firing element involving an external or terminal transition, then for a firing element involving an internal transition (not including terminal transitions), and finally for a set of firing elements in general. (Recall that the functions *Term*, *Trivial* and *FiringElements* have been introduced at the start of Section 7.4.9.)

Proposition 7.29.

a. $(t,c) \in FE|_{ETT}$ is live $\Leftrightarrow$

$$\Big(\forall scc \in Term(SCC_{\mathcal{G}}) : (t,c) \in FiringElements(scc)\Big) \; \wedge$$
$$\Big(\forall v \in \mathcal{V}_{\mathcal{G}} : \forall M \in [[v^{-t}\rangle : \forall x'' \in X'' : (M|_{x''})^c \in Term(SCC_{x''})$$
$$\Rightarrow \exists (v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \; : \; \exists M_1 \in [_{t_1}[v^{-t}\rangle \; : \; M_1|_{^\circ t_1{}^\circ} = m_1\Big)$$

b. $(t,c) \in FE|_{T-ETT}$, where $t \in N_{x''}$ for some $x'' \in X''$ is live $\Leftrightarrow$

$$\Big(\forall scc \in Term(SCC_{\mathcal{G}}) : \exists \, v \in scc : (t,c) \in FiringElements([[v^{-t}\rangle)\Big) \; \wedge$$
$$\Big(\forall v \in \mathcal{V}_{\mathcal{G}} : \forall M \in [[v^{-t}\rangle : (M|_{x''})^c \in Term(SCC_{x''})$$
$$\Rightarrow ((t,c) \in FiringElements([[M\rangle) \; \vee$$
$$\exists (v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \; : \; \exists M_1 \in [_{t_1}[v^{-t}\rangle \; : \; M_1|_{^\circ t_1{}^\circ} = m_1\Big)$$

c. $X \subseteq FE$ where $t \in N_{x''}$ for some $x'' \in X''$is live $\Leftrightarrow$

$$\Big(\forall scc \in Term(SCC_{\mathcal{G}}) : (X \cap FiringElements(scc) \neq \emptyset) \; \vee$$
$$\big(\exists \, v \in scc : X \cap FiringElements([[v^{-t}\rangle) \neq \emptyset\big)\Big) \; \wedge$$
$$\Big(\forall v \in \mathcal{V}_{\mathcal{G}} : \forall M \in [[v^{-t}\rangle : (\forall x'' \in X'' : (M|_{x''})^c \in Term(SCC_{x''}))$$
$$\Rightarrow \big(X \cap FiringElements([[M\rangle) \neq \emptyset\big) \; \vee$$
$$\exists (v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} \; : \; \exists M_1 \in [_{t_1}[v^{-t}\rangle \; : \; M_1|_{^\circ t_1{}^\circ} = m_1\Big)$$

cont'd ...

Explanation:

a. A firing element involving an external or terminal transition is live iff it occurs in all terminal SCCs of the global graph $\left(\forall scc \in Term(SCC_G) : (t,c) \in FiringElements(scc)\right)$, and it is always possible to reach a marking where an external or terminal transition is enabled.

b. A firing element (t,c) internal to the supernode x'' is live iff for all terminal SCCs of the global graph it is possible to reach a marking where (t,c) is enabled $\left(\forall scc \in Term(SCC_G) : \exists\, v \in scc : (t,c) \in FiringElements([[v^{-t}\rangle])\right)$, and it is always possible to reach a marking where (t,c) is enabled or a marking where an external or terminal transition is enabled.

c. A set of firing elements X is live iff for each terminal SCC of the global graph there is an occurrence of some element of X $\left(\forall scc \in Term(SCC_G) : (X \cap FiringElements(scc) \neq \emptyset\right)$, or there is an internally reachable marking where a transition in X is enabled $\left(\exists\, v \in scc : X \cap FiringElements([[v^{-t}\rangle]) \neq \emptyset\right)$. Furthermore it always possible to reach a marking where an element in X is enabled or a marking where an external or terminal transition is enabled.

Proof.
(a) and (b) are special cases of (c), so we only prove (c).
($\Rightarrow$) Let X be a live set of firing elements.
Here we show the first conjunct of the right hand side holds. Let *scc* be a terminal SCC of the global graph, and M be a marking such that M^t is a vertex of *scc*. From a global vertex in a terminal SCC of the global graph it is only possible to reach other global vertices in the SCC. Therefore the only firing elements enabled at any marking reached from M are those that lead to a marking corresponding to a global vertex in *scc*, together with those enabled at markings internally reachable from a global vertex in *scc*. That is, the only firing elements that can become enabled from M are those in $FiringElements(scc)$ together with those in $FiringElements([[v^{-t}\rangle])$ for $v \in scc$. Since X is live then from M it is possible to reach a marking where one of the firing elements of X is enabled. Therefore $(X \cap FiringElements(scc) \neq \emptyset) \vee (\exists\, v \in scc : X \cap FiringElements([[v^{-t}\rangle]) \neq \emptyset)$.

Now we show the second conjunct of the right hand side holds. Consider a marking M internally reachable from a global vertex, such that for all $x'' \in X''$ the marking $M|_{x''}$ is in a terminal SCC of the supernode graph of x''. The markings that can be reached from M are those that can be internally reached together with those that can be reached by the occurrence of an external or terminal transition from one of these internally reachable markings (possibly followed by the occurrence of further internal and external transitions). Therefore the firing elements of X enabled in markings reachable from M are $FiringElements([[M\rangle])$ together with those enabled at a marking reached by the occurrence of an external or terminal transition from one of these internally reachable markings (possibly followed by the occurrence of further internal and external transitions). Since X is live, either X contains a firing element enabled at a marking internally reachable from M, or there is an external or terminal transition enabled at a marking internally reachable from M. Proposition 7.23 tells us that for the occurrence of an external or terminal transition $t_1 \in ETT$ from any marking M_1 then there is an edge

$(v_1,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}}$ such that $M_1 \in [_{t_1}[v_1^{-\ell}\rangle$ and $M_1|_{t_1}^{\circ} = m_1$. So we have $(X \cap \mathit{FiringElements}([[M\rangle) \neq \emptyset) \ \vee \ \exists\,(v,(m_1,(t_1,c_1),m_2),v_2) \in \mathcal{E}_{\mathcal{G}} : \exists\, M_1 \in [_{t_1}[v^{-\ell}\rangle : M_1|_{t_1}^{\circ} = m_1$, as required.

($\Leftarrow$) By Lemma 7.22 every marking is internally reachable from a global vertex. Consider some marking M internally reachable from a global vertex $v_1 \in \mathcal{V}_{\mathcal{G}}$. From M it is possible to internally reach a marking M_1, where for each $x'' \in X''$ the marking $M_1|_{x''}$ is in a terminal SCC of the supernode graph of x''. By the second conjunct of the condition we are guaranteed that an element in X is enabled at a marking internally reachable from M_1, or that an external or terminal transition is enabled from a marking internally reachable from M_1. By the occurrence of an external or terminal transition we can obtain another marking, M_2, corresponding to another global vertex, $v_2 \in \mathcal{V}_{\mathcal{G}}$. Since v_2 is reached by the occurrence of an external or terminal transition then $v_2 \neq v_1$. From the global vertex v_2 it is possible to reach a marking corresponding to a global vertex in a terminal SCC of the global graph. From the first conjunct of the condition, one of its successors enables a firing element in X. ◊

Sketch of algorithm: Given a set of transitions, X, we want to check if it is live. We mark the vertices in each terminal SCC of the global graph, *scc*, such that there is a firing element of X in the transitions of the SCC (i.e. in $\mathit{FiringElements}(scc)$) or there is a firing element of X internally reachable from a global vertex v in *scc*. The marked vertices are those that satisfy the first conjunct of the condition on the right hand side of (c). Now if there exists a terminal SCC of the global graph without any marked vertex, the first conjunct is not satisfied, thus X is not live. Otherwise, we have to check the second conjunct of the condition. For each vertex in the global graph we consider the internally reached markings such that every supernode marking is in a terminal SCC of the corresponding supernode graph. For each such marking we check if it is possible to reach an enabled firing element involving an external transition, terminal transition, or a firing element in X. If one of these markings does not satisfy this requirement then X is not live, otherwise it is.

Example: As an example we check whether the set of all firing elements of the net of Figure 7.3 is live. The terminal SCCs of the global graph are the vertices v_3 and v_4 of Figure 7.4 (c). There are no external or terminal transitions enabled from these vertices. From each of these vertices a transition internal to the superplace of p_2 is enabled, and so we mark the global vertices v_3 and v_4. Since each terminal SCC of the global graph has a marked vertex then the first conjunct is satisfied. We now check the second conjunct of the condition. For each vertex in the global graph we consider the internally reached markings such that every supernode marking is in a terminal SCC of the corresponding supernode graph. For each of these markings it is possible to reach an enabled internal transition or an enabled external transition and so the second conjunct of the condition is also satisfied. Thus the set of all firing elements is live.

7.4.14 Boundedness

Intuitively, upper and lower bounds tell us how many and how few token elements we can get. Given a place $p \in P$:

$$BestUpperBound(p) = max_{M \in [M_0\rangle}\left(M|_p\right)$$

$$BestLowerBound(p) = min_{M \in [M_0\rangle}\left(M|_p\right)$$

Here we explain how boundedness properties can be checked using a RNSS without unfolding. We use the property that all places are either external, or internal to a particular supernode.

Proposition 7.30. Given a place $p \in P$:

a. $$BestUpperBound(p) = \begin{cases} max_{M \in \mathcal{V}_{x''}}\left(M|_p\right) & \text{,if } \exists x'' \in X'' \,:\, p \in P_{x''} \\ max_{v \in \mathcal{V}_G}\left(v|_p\right) & \text{,otherwise} \end{cases}$$

b. $$BestLowerBound(p) = \begin{cases} min_{M \in \mathcal{V}_{x''}}\left(M|_p\right) & \text{,if } \exists x'' \in X'' \,:\, p \in P_{x''} \\ min_{v \in \mathcal{V}_G}\left(v|_p\right) & \text{,otherwise} \end{cases}$$

Explanation:
If the place is internal to a supernode, the best upper bound can be found directly in the supernode graph. Otherwise the best upper bound can be found directly from the global graph. Similarly for the best lower bound.

Proof.

a. By Definition 7.16 the RNSS contains the restriction to the supernode of x'' of every marking internally reachable from a global vertex. By Lemma 7.22 all markings are internally reachable from a global vertex. Therefore the best upper bound of each place internal to a supernode can be found from the graph of the supernode, and the best upper bound of an external place can be found by examining the global graph.

b. Similar to (a).

◊

Sketch of algorithm: The bounds of a single place can easily be checked by either examining the vertices of a given supernode graph (if the place is internal to that supernode), or by examining the vertices of the global graph (if the place is an external place).

Example: By examining the vertices of the supernode graph of p_2 we can show that the place p_2-*buf* contains at most one token.

Note: The bounds of a single place can be checked efficiently because they only require examining the vertices of a supernode graph, or the global graph. Clearly the above propositions can be generalised for the bounds of several places rather than one.

7.4.15 An Optimisation to the RNSS Algorithm

In this section we consider an optimisation to the RNSS algorithm. Recall from Definition 4.6 that a sequence is *complete* if all border transitions occur with matching modes, and that every complete sequence of the refined net has a corresponding abstract sequence. The edges of the global graph of the RNSS correspond to the occurrence of an external or terminal transition, where terminal transitions are defined to be the output border transitions of a supertransition. This means that a complete firing sequence of a supertransition may result in several edges in the global graph of the RNSS — one for each occurrence of an output border transition. For example, in the net of Figure 7.5 a successor global vertex, say v_1, is added for the occurrence of *out1*. From v_1 a successor is added to the global graph for the occurrence of *out2*. While this may seem counterintuitive, as we saw in Section 7.4.7 it allows the RNSS to be unfolded to the full reachability graph.

The optimisation we consider here is to only add a successor to the global graph once all output border transitions of a supertransition have occurred with matching mode, rather than for the occurrence of each output border transition of the supertransition. For example, in the net of Figure 7.5 an edge will only be added to the global graph once both *out1* and *out2* have occurred with matching mode. Such an optimisation may significantly improve the performance of the RNSS algorithm since it reduces the number of global vertices and global edges.

To allow us to easily detect sequences where all output border transitions have occurred, we add to the canonical basis a place for each output border transition, and a transition named *finish*. As shown in Figure 7.5 the place added for each output border transition indicates that the output border transition has fired. The *finish* transition is enabled once all output border transitions have fired with a given mode.

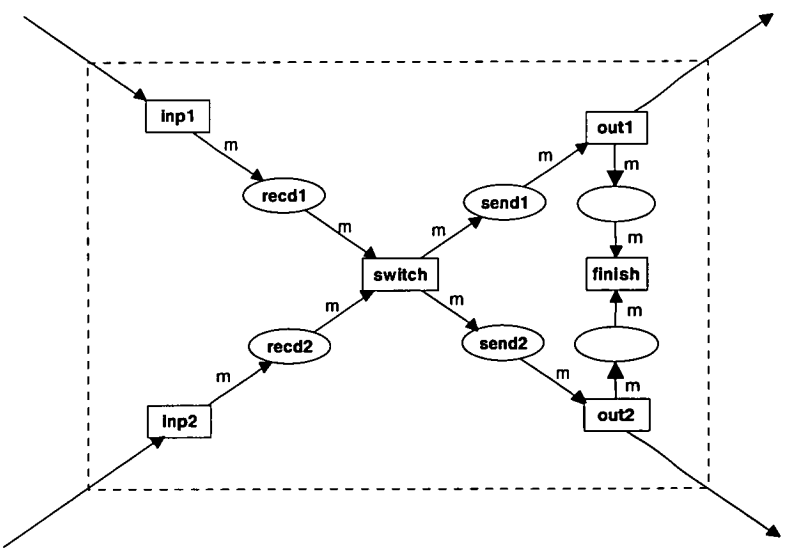


Figure 7.5: *finish* transition added to the canonical basis of a supertransition

The optimised RNSS definition, the definition for unfolding the RNSS, and the RNSS algorithm are exactly the same as that of the standard RNSS (as presented in Sections 7.4.1 – 7.4.8), except that the *terminal* function of Definition 4.3 is redefined so that it returns the *finish* transition of the supertransition rather than the output border transitions of the supertransition. In other words, we add an edge to the RNSS due to the occurrence of the *finish* transition rather than the occurrence of an output border transition. In the remainder of this section, unless otherwise specified, we will assume this new definition of the *terminal* function. Further to this, when we refer to a definition we mean the definition modified to use this redefined function. For example, if we refer to the definition of the RNSS, Definition 7.16, we mean Definition 7.16 with the redefined *terminal* function. It is important to note that we only use the definitions from the previous sections with the redefined *terminal* function, not the lemmas and propositions. This is simply to save re-writing the definitions. Propositions involving the optimised RNSS are proved in this section.

The unfolded optimised RNSS is not guaranteed to be isomorphic to the full reachability graph. However, under certain conditions the unfolded optimised RNSS can be guaranteed to contain the same dead markings as the full reachability graph. (We note that one of the most important reachability problems is finding dead markings because a dead marking often means a severe design or modelling error or is a result of a terminating computation.)

In the remainder of this section we first propose a condition under which the unfolded optimised RNSS contains all reachable dead markings of the full reachability graph, and further that all dead markings of the unfolded optimised RNSS are reachable dead markings in the full reachability graph. We then prove that this property holds. The basic idea of the proof is to show that under the required conditions the unfolded optimised RNSS has the same dead markings as a stubborn set reduced reachability graph. The property follows since every stubborn set reduced reachability graph has the same dead markings as the full reachability graph [194]. In the following, Definitions 7.31 and 7.32, and Proposition 7.33 are due to Lakos [121].

In some situations the unfolded optimised RNSS will miss important activity of the state space. For example, if an output border transition can occur, but following this the *finish* transition never occurs then only adding successors to the global graph following occurrences of the *finish* transition will mean that the states due to the occurrence of the border transition will be missed. So that important activity of the state space is not missed, what we require is that once the *switch* transition occurs (with a given mode) then the *finish* transition is guaranteed to occur (with that mode), regardless of the further occurrence of input border transitions. We say such a *finish* transition of the supertransition is *locally live*. This is formalised in Definition 7.31.

Definition 7.31. In this definition we assume that all markings and steps are local to a supertransition of $t'' \in T''$ and that all steps are singleton steps. For $Y_a^*, Y_b^* \in \sigma\mathbb{Y}$ we use $Y_a^* \mid\mid Y_b^*$ to denote an interleaving of the sequences Y_a^* and Y_b^*. We refer to *dependent transitions* of a given transition in a sequence. These are the transitions that would be disabled except for the preceding transitions.
Given a local sequence $Y_0^* \in \sigma\mathbb{Y}$ such that $M_0[Y_0^*(t_{sw}, c))M_{sw}$, where t_{sw} is the *switch* transition of the supertransition of t'' then the *finish* transition, t_{fin}, of the supertransition of t'' is *locally live* if:

a. for all $Y_1^* \in \sigma\mathbb{Y}$ of a supertransition such that $M_{sw}[Y_1^*\rangle$ there exists $Y_2^* \in \sigma\mathbb{Y}$ such that $M_{sw}[Y_1^*Y_2^*\rangle$ and (t_{fin}, c) appears in $Y_1^*Y_2^*$

b. for all such $Y_1^*Y_2^*$ (satisfying the condition in part (a)), there is a reordered sequence $Y_3^*Y_4^*$ such that $M_{sw}[(t_{sw}, c)Y_3^*(t_{fin}, c)Y_4^*\rangle$ and no transition in $inpbdr(N_{t''}) \cup \{t_{sw}\}$ occurs in Y_3^*. Note that the reordering means that $Y_1^*Y_2^* = Y_3^*(t_{fin}, c) \mid\mid Y_4^*$. Further, it must be possible to reorder the sequence one step at a time, i.e. there are a number of sequences Y_{3i}^*, Y_{5i}^*, $Y_{4i}^*(i = 0 \ldots n)$ which satisfy the following conditions:

i. $M_{sw}[Y_{3i}^*Y_{5i}^*Y_{4i}^*\rangle$
ii. $Y_1^*Y_2^* = Y_{3i}^* \mid\mid (Y_{5i}^*Y_{4i}^*)$
iii. $Y_{4i}^* = tail(Y_{4(i-1)}^*)$, for $i > 0$
iv. $Y_4^* = Y_{5n}^*Y_{4n}^*$
v. $\#(Y_{30}^*) = \#(Y_{50}^*) = 0$
vi. $Y_{3i}^* = Y_{3(i-1)}^*$ or $Y_{3i}^* = Y_{3(i-1)}^*(head(Y_{4(i-1)}^*))$, for $i > 0$
vii. $Y_{5i}^* = Y_{5(i-1)}^*$ or $Y_{5i}^* = Y_{5(i-1)}^*(head(Y_{4(i-1)}^*))$, for $i > 0$
viii. $Y_{3n}^* = Y_3^*(t_{fin}, c)$

c. for all such Y_3^*, Y_4^* (satisfying the condition of part (b)), the possible occurrence of a sequence including an input border transition or *switch* transition of the supertransition plus dependent transitions, say Y_5^*, does not affect the enabling of Y_3^*, that is $M_{sw}[Y_{31}^*Y_5^*Y_{32}^*(t_{fin}, c))\rangle$ where $Y_{31}^*Y_{32}^* = Y_3^*$.

Note:

a. Once the *switch* transition occurs, the *finish* transition can occur. In other words all enabled (internal) sequences that follow the *switch* transition can be extended to a sequence that involves the *finish* transition.

b. Every internal sequence leading to the occurrence of the *finish* transition can be reordered to a sequence where the border input transitions and *switch* transition do not occur before the *finish* transition has occurred. Further this reordering can be done one step at a time — elements of Y_4 are progressively appended to Y_3 or Y_5.

c. An enabled sequence involving an input border transition of the supertransition or the *switch* transition of the supertransition can be inserted without affecting the remainder of the sequence.

For a net where the *finish* transition of each supertransition is locally live then we show that the unfolded optimised RNSS contains the same dead markings as the full reachability graph. To do this we first show that the *finish* transition is locally live if and only if it is *stubbornly live*, as defined in Definition 7.32. (Recall from Chapter 6 that the dynamically stubborn set definition gives the basic criteria for stubbornness and that various static definitions have been proposed that ensure the resulting stubborn set is dynamically stubborn. Unless otherwise stated, when we refer to stubborn sets in this section we mean dynamic stubborn sets.)

Definition 7.32. In this definition we assume that the markings and steps are all local to a supertransition $t'' \in T''$ and that all steps are singleton steps (since singleton steps are usual for stubborn set analysis).
Given a local sequence $Y_0{}^* \in \sigma\mathbb{Y}$ such that $M_0[Y_0{}^*(t_{sw},c)\rangle M_{sw}$, where t_{sw} is the *switch* transition of the supertransition of t'' then the *finish* transition, t_{fin}, of the supertransition of t'' is *stubbornly live* if:

a. for all $Y_1{}^* \in \sigma\mathbb{Y}$ such that $M_{sw}[Y_1{}^*\rangle$ there exists $Y_2{}^* \in \sigma\mathbb{Y}$ such that $M_{sw}[Y_1{}^*Y_2{}^*\rangle$ and (t_{fin},c) occurs in $Y_1{}^*Y_2{}^*$

b. for all such $Y_1{}^*Y_2{}^*$ (satisfying the condition in part (a)), there is a reordered sequence $Y_3{}^*Y_4{}^*$ such that $M_{sw}[Y_3{}^*(t_{fin},c)Y_4{}^*\rangle$ and for every step in $Y_3{}^*$, viz. Y_6 in $Y_5{}^*Y_6Y_7{}^* = Y_3{}^*(t_{fin},c)$, with $M_{sw}[Y_5{}^*\rangle M_S[Y_6Y_7{}^*\rangle$ there is a stubborn set S at M_S such that $Y_6 \in S$ and $(\forall(t,c) \in S : t \notin inpbdr(N_{t''}) \cup \{t_{sw}\})$ (Recall that all steps are singleton and so Y_6 is a single firing element.)

Note:

a. All enabled (internal) sequences that follow the *switch* transition can be extended to a sequence that involves the *finish* transition.

b. There is a reordered sequence such that at each marking M_S reachable from M_{sw} there is a stubborn set S at M_S such that the next firing element in the sequence is in S and S does not include any firing elements involving input border transitions or the *switch* transition.

The following proposition tells us that the local liveness property is equivalent to a stubbornness property. Therefore we can use stubbornness to verify local liveness on-the-fly.

Proposition 7.33. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, and $t'' \in T''$, then the *finish* transition of the supertransition of t'' is *locally live* iff it is *stubbornly live.*

Proof. Clearly part (a) follows directly in both directions, so we just focus on the remainder. Given a local sequence $Y_0{}^* \in \sigma\mathbb{Y}$ such that $M_0[Y_0{}^*(t_{sw},c)\rangle M_{sw}$, where t_{sw} is the *switch* transition of t'' then:
($\Rightarrow$) Suppose that we have local liveness. Consider the following conditions:

1. $head(Y_{4(i-1)}{}^*)$ is not an input border transition or *switch* transition

2. $M_{sw}[Y_{3(i-1)}{}^*\rangle M[head(Y^*_{4(i-1)})\rangle$ and there is a stubborn set at the marking M including $head(Y^*_{4(i-1)})$ but excluding all the elements of $Y_{5(i-1)}{}^*$

We define the reordering of the sequence as specified in Definition 7.31 (b) as follows: if the above two conditions hold then $Y_{3i}{}^* = Y_{3(i-1)}{}^* head(Y^*_{4(i-1)})$ and $Y_{5i}{}^* = Y_{5(i-1)}{}^*$ otherwise $Y_{3i}{}^* = Y_{3(i-1)}{}^*$ and $Y_{5i}{}^* = Y_{5(i-1)}{}^* head(Y^*_{4(i-1)})$. Note that, under this criterion, if $head(Y_{4(i-1)}{}^*)$ is appended to $Y_{5(i-1)}{}^*$, then appending it to $Y_{3(i-1)}{}^*$ is not an option since the stubborn set identified in condition 2 will include elements of $Y^*_{5(i-1)}$, and therefore Definition 7.31 (c) will not hold. Also note that this approach maximises the length of $Y_{3i}{}^*$ and thus minimises the possible conflicts of remaining steps of $Y_{4i}{}^*$ with those in $Y_{5i}{}^*$. Therefore, if it is possible for (t_{fin}, c) to be appended to $Y_{3i}{}^*$ (as required by Definition 7.31 (b) viii), then this process will allow it to happen.

Now suppose that there is a reordered sequence $Y_3{}^*$ as defined above, and a step Y_7 in $Y_6{}^* Y_7 Y_8{}^* = Y_3{}^*(t_{fin}, c)$, where $M_{sw}[Y_6{}^*\rangle M_S[Y_7 Y_8{}^*\rangle$ such that every stubborn set at M_S that includes Y_7 also includes an input border transition or *switch* transition. This means that there is a step sequence enabled at M_S, say $Y_9{}^*$ which includes the occurrence of an input border transition or *switch* transition and which disables Y_7. Without loss of generality, $Y_9{}^*$ starts with a firing element involving an input border transition or the *switch* transition, and the steps following this firing element are initially disabled. If it does not, consider $Y_6{}^*$ as now extended by the sequence preceding the first input border transition, and consider the first enabled firing element following the input border transition or *switch* as a new Y_7 and the new sequence $Y_6{}^*$ can be completed under part (a) with a new $Y_8{}^*$ (and the original assumptions still hold). Now $Y_9{}^*$ violates part (c) of Definition 7.31. The desired property is thus proved by contradiction.

($\Leftarrow$) Now assume that the stubborn set property holds, that is Definition 7.32 holds. Consider a sequence $Y_5{}^* Y_6 Y_7{}^*$ such that $M_{sw}[Y_5{}^*\rangle M_S[Y_6 Y_7{}^*\rangle$. By Definition 7.32 (b) there is a stubborn set S at M_S such that Y_6 is in S and S does not include any firing elements involving input border transitions or the *switch* transitions.

Suppose that no input border transition or *switch* can be enabled between (t_{sw}, c) and (t_{fin}, c). Then the stubborn set property holds trivially as do the properties of Definition 7.31 (b) and (c). Now suppose that we insert before Y_6 a sequence $Y_8{}^*$ involving an input border transition or *switch* transition plus dependent transitions. Definition 7.32 (b) implies all the elements of $Y_8{}^*$ must be outside the stubborn set S or else, since they are dependent on the input border transition or *switch* transition, their inclusion would require the inclusion of the input border transition or *switch* transition. Therefore we can reorder $Y_8{}^* Y_6 Y_7{}^*$ to $Y_6 Y_8{}^* Y_7{}^*$. The same arguments apply to the elements of $Y_7{}^*$ and so we can reorder $Y_6 Y_8{}^* Y_7{}^*$ to $Y_6 Y_7{}^* Y_8{}^*$ without affecting the enabling of the sequence. Thus, property (c) of Definition 7.31 holds. This applies no matter how long a prefix of $Y_8{}^*$ we consider and hence we also have Definition 7.31 (b). This holds independent of the particular Y_6 chosen. Hence the locally live property holds. $\Diamond$

A reduced graph can be constructed using the stubborn sets identified in Definition 7.32. We refer to such a reduced graph as a *stubbornly live reduced graph.* To construct a stubbornly live reduced graph, when a *switch* transition has occurred and the corresponding *finish* transition has not yet occurred then only firing elements in the stubborn set identified in Definition 7.32 (b) are considered, otherwise all enabled firing elements are considered. In other words, suppose the marking M_s is reached following the occurrence of a *switch*

transition of a supertransition, but before the corresponding occurrence of the *finish* transition. The immediate successors from M_s, are obtained by the occurrence of firing elements from a stubborn set S. The stubborn set S only contains firing elements local to the supertransition and does not include input border transitions or the *switch* transition. Since input border transitions of the supertransition are not included in S then the transitions of S are independent of the transitions external to the supertransition. Therefore the local stubborn set S is also a stubborn set for the global net, and (hence) a stubbornly live reduced graph is a stubborn set reduced graph.

We now prove that if the net is such that the *finish* transition of each supertransition is stubbornly live (or equivalently, locally live) then the unfolded optimised RNSS contains the same dead markings as a stubbornly live reduced graph. It is useful to first prove Lemmas 7.34 – 7.36. Lemma 7.34 says that if M is reachable from a marking represented by a global vertex of the optimised RNSS by a sequence that does not include the occurrence of a *switch* transition then M is internally reachable from a global vertex. Lemma 7.35 tells us that any marking of a stubbornly live reduced graph is also a marking of the unfolded optimised RNSS. Lemma 7.36 says that any marking internally reachable from a global vertex of the optimised RNSS is a reachable marking.

Lemma 7.34. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \rightarrow N'$, the optimised RNSS $(\mathcal{G}, \mathcal{G}_{X''})$, a marking M_1 such that $M_1^{\natural} \in \mathcal{V}_{\mathcal{G}}$ and a sequence $Y^* \in \sigma\mathbb{Y}$ such that $M_1[Y^*\rangle M_2$ and no *switch* transition occurs in Y^* then there exists $v \in \mathcal{V}_{\mathcal{G}}$ such that $M_2 \in [[v^{-\natural}\rangle$.

Proof. We show that Y^* can be reordered so that the marking following each occurrence of an external transition corresponds to a global vertex. Since there is no *switch* transition in Y^* then Y^* does not contain the occurrence of any terminal transitions of a supertransition. Further the optimised RNSS is the same as the standard RNSS before the occurrence of a *switch* transition. Therefore the reordering follows using the same arguments as in the proof of Lemma 7.22 where the case concerning the occurrence of terminal transitions can be safely ignored. The required result follows from this reordering, since M_2 is internally reachable from the global vertex corresponding to the marking following the last external transition in Y^*. ◊

Lemma 7.35. Given an abstract net N', and a refined net N related to N' by a system morphism $\phi : N \rightarrow N'$, the optimised RNSS $(\mathcal{G}, \mathcal{G}_{X''})$, an unfolding of the optimised RNSS, $(\mathcal{V}_u, \mathcal{E}_u)$, and a stubbornly live reduced graph $(\mathcal{V}_s, \mathcal{E}_s)$ then:

$$M \in V_s \Rightarrow M \in V_u$$

Proof. Since $M \in V_s$ then we know M is a reachable marking (i.e. $M \in \mathbb{M}_R$). Therefore there exists a step sequence $Y^* = (t_1, c_1) \ldots (t_m, c_m) \in \sigma\mathbb{Y}$, such that $M_0[Y^*\rangle M$.
We now show using induction that regardless of the number of *switch* transitions that occur in Y^* then $M \in V_u$.

Inductive Proposition: If M is reachable by a sequence Y^* such that the number of *switch* transitions in Y^* is n then $M \in V_u$.

Basis: We first show the inductive proposition holds for $n = 0$.
Since Y^* does not contain any *switch* transition then the optimised RNSS is the same as the standard RNSS. It therefore follows from Proposition 7.23 that $M \in V_u$.

Inductive Assumption: Assume that when $n = k$ then $M \in V_s \Rightarrow M \in V_u$

Inductive Step: Consider the case where $n = k + 1$. We know from the inductive assumption that the marking M_k following the k-th *finish* transition is in the unfolded graph. Since a global vertex is added to the RNSS following each occurrence of a *finish* transition then $v_1 = M_k{}^t \in \mathcal{V}_\mathcal{G}$. Suppose M_1 is the marking immediately preceding the $(k+1)$-st *switch* transition. Since M_1 is reachable from M_k by a sequence that does not involve a *switch* transition then it follows from Lemma 7.34 that there is a global vertex $v_2 \in \mathcal{V}_\mathcal{G}$ such that $M_1 \in [[v_2^{-t}\rangle$.

Now, since the *finish* transition is live then there are two cases:

i. M is reached before the *finish* transition fires.
In this case the stubbornly live graph construction implies that M is internally reachable in the supertransition of t''. Therefore M is internally reachable from v_2.

ii. M is reached after the *finish* transition is fired.
Suppose M_{fin} is the marking immediately following the *finish* transition. Since the optimised RNSS adds a successor to the global graph following the occurrence of a *finish* transition then $v_3 = M_{fin}{}^t \in \mathcal{V}_\mathcal{G}$. Since there are no further occurrences of a *switch* transition in the sequence from M_{fin} to M then by Lemma 7.34 there is a global vertex from which M is internally reachable.

We have shown that in all cases M is internally reachable from a global vertex. Since the unfolding of the optimised RNSS adds a vertex to the unfolded graph for every marking internally reachable from a global vertex then M is in the unfolded graph.

Conclusion: The inductive proposition holds for $n = 0$ and if we assume $n = k$ then it holds for $n = k + 1$ therefore by the principle of mathematical induction the proposition holds.

◊

Lemma 7.36. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, the optimised RNSS $(\mathcal{G}, \mathcal{G}_{X''})$ then:

$$\exists\, v \in \mathcal{V}_\mathcal{G} \;:\; M \in [[v^{-t}\rangle \Rightarrow M \in \mathbb{M}_R$$

Proof. The required result holds using the same arguments as in the ($\Leftarrow$) case of Proposition 7.22. (Note that the arguments remain valid for the optimised RNSS.) ◊

We now prove that the dead markings of the unfolded optimised RNSS are the same as those of a stubborn set reduced graph.

Proposition 7.37. Given an abstract net N', a refined net N related to N' by a system morphism $\phi : N \to N'$, the optimised RNSS $(\mathcal{G}, \mathcal{G}_{X''})$, an unfolding of the optimised RNSS, $\mathcal{G}_u = (\mathcal{V}_u, \mathcal{E}_u)$, and a stubbornly live reduced graph, $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$, then:

$$M \text{ dead in } \mathcal{G}_s \Leftrightarrow M \text{ dead in } \mathcal{G}_u$$

Proof. ($\Rightarrow$) Given M is dead in $\mathcal{G}_s$
By Lemma 7.35 we know that $M \in \mathcal{V}_u$. Suppose that M is not dead in $\mathcal{G}_u$. Therefore there exists $(M_1,(t,c),M_2) \in \mathcal{E}_u$. By Definition 7.17 this implies either:

i. there exists $(M_1,(t,c),M_2) \in \mathcal{E}_1$ of Definition 7.17.
$\Rightarrow \exists\, (v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_{\mathcal{G}}$ of Definition 7.16, where $M_1 \in [[v_1^{-t}\rangle$ and $M_2 \in [[v_2^{-t}\rangle$ and $M_1|_{t^\circ} = m_1$ and $M_2|_{t^\circ} = m_2$, and $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$.
Since $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_{\mathcal{G}}$ then by Definition 7.16 there is a marking, say M_1', reachable from v_1^{-t} by transitions local only to supernodes that affect the enabling of t such that (t,c) is enabled at M_1', and the marking of the supernodes adjacent to t in M_1' is equal to m_1. Further the occurrence of (t,c) at M_1' leads to some marking $M_2' \in v_2^{-t}$, and the marking of the supernodes adjacent to t in M_2' is equal to m_2. In other words, by Definition 7.16 there exists $M_1' \in [_t[v_1^{-t}\rangle$ and $M_2' \in v_2^{-t}$ where $m_1 = M_1'|_{t^\circ}$ and $m_2 = M_2'|_{t^\circ}$ and $M_1'[(t,c)\rangle M_2'$. Since $M_1'[(t,c)\rangle M_2'$ and $M_1'|_{t^\circ} = m_1 = M_1|_{t^\circ}$ and $M_2'|_{t^\circ} = m_2 = M_2|_{t^\circ}$, and $M_1|_{(P-{}^\circ t^\circ)} = M_2|_{(P-{}^\circ t^\circ)}$ then $M_1[(t,c)\rangle M_2$. Further, since M_1 is internally reachable from the global vertex v_1 respectively, then Lemma 7.36 implies M_1 is reachable.
Therefore $(M_1,(t,c),M_2)$ is an edge of the full reachability graph. However, all deadlock states of a stubborn set reduced graph are known to be deadlock states of the full reachability graph. Therefore we have a contradiction.

or

ii. there exists $(M_1,(t,c),M_2) \in \mathcal{E}_2$ of Definition 7.17.
$\Rightarrow \exists\, (m_1,(t,c),m_2) \in \mathcal{E}_{x''}$ of Definition 7.16 for some $x'' \in X''$ such that $M_1|_{x''} = m_1$, and $M_2|_{x''} = m_2$ and $M_2 = M_1 - m_1 + m_2$, where M_1 and M_2 are internally reachable from a global vertex. Since M_1 and M_2 are internally reachable from a global vertex then Lemma 7.36 implies M_1 and M_2 are reachable. Since $M_2 = M_1 - m_1 + m_2$ and $m_1[(t,c)\rangle m_2$ then $M_1[(t,c)\rangle M_2$. Therefore $(M_1,(t,c),M_2)$ is an edge of the full reachability graph. Again we have a contradiction for the same reason as in case (i) above.

In both cases we have a contradiction. Thus the required result is proved by contradiction.

($\Leftarrow$) Now suppose M is dead in $\mathcal{G}_u$ but not dead in the full reachability graph.
Since $M \in \mathcal{V}_u$ then by Definition 7.17 there exists $v_1 \in \mathcal{V}_{\mathcal{G}}$ such that $M \in [[v_1^{-t}\rangle$. Therefore by Lemma 7.36 M is a reachable marking. Since M is not dead in the full reachability graph then there exists $(t,c) \in FE$ such that $M[(t,c)\rangle$. Now there are two cases:

i. t is an external transition or terminal (*finish*) transition
In this case since $M \in [[v_1^{-t}\rangle$ and $M[(t,c)\rangle$, then there exists $M' \in [_t[v_1{}^{-t}\rangle$ such that $M'|_{t^\circ}{}^\circ = M|_{t^\circ}{}^\circ$ and $M'[(t,c)\rangle$. Therefore there is an edge $(v_1,(m_1,(t,c),m_2),v_2) \in \mathcal{E}_1$ of Definition 7.16. But since $M \in [[v_1^{-t}\rangle$ and $M|_{t^\circ}{}^\circ = M'|_{t^\circ}{}^\circ = m_1$ and $M'[(t,c)\rangle$ then Definition 7.17 implies M is not dead in $\mathcal{G}_u$. This is a contradiction.

or

ii. t is internal to a supernode and is not a terminal (*finish*) transition. In this case since $M[(t,c)\rangle$ then there is a supernode $x'' \in X''$ such that $t \in T_{x''}$ and $(M|_{x''},(t,c),M_2|_{x''}) \in \mathcal{E}_{x''}$. Definition 7.17 then implies M is not dead in $\mathcal{G}_u$, which is a contradiction.

In both cases we have a contradiction. Therefore M is a reachable dead marking of the full reachability graph. Since $\mathcal{G}_s$ is a stubborn set reduced graph then it contains all reachable dead markings [194]. Hence M is in $\mathcal{G}_s$ and furthermore M is dead in $\mathcal{G}_s$. ◊

Since the dead markings of a stubborn set reduced graph and the full reachability graph are the same [194] then Proposition 7.37 tells us that if each *finish* transition is locally live (or, equivalently, stubbornly live) the dead markings of the unfolded optimised RNSS are the dead markings of the full reachability graph. Further, since the dead markings of the unfolded optimised RNSS are the dead markings of the full reachability graph then the arguments of the proof of Proposition 7.27 remain valid for the optimised RNSS, and hence the dead markings of the optimised RNSS can be found without unfolding using Proposition 7.27. The problem remaining is how to show or ensure that a *finish* transition is locally live (or, equivalently, stubbornly live).

One approach to showing that the *finish* transition is locally live is to construct the reachability graph of the supertransition, and at each M_S as defined in Definition 7.32 (b) search for a dynamic stubborn set satisfying the conditions of Definition 7.32 (b). The *finish* transition is locally live if and only if such a dynamic stubborn set exists. It is also possible to instead search for a static stubborn set. Since each static stubborn set is a dynamic stubborn set [194] then if at each state M_S there is a static stubborn set that meets the requirements of Definition 7.32 (b) then *finish* transition is locally live.

An alternative approach is to impose a static condition on the supertransition. One such condition is to require a semaphore place in the canonical basis of the supertransition, as shown in Figure 7.6. (In this figure the components added to the basis have been rendered using thicker lines.) The newly added place — the *semaphore* place — contains a single token. This place is accessed as a side condition of each input border transition. The token in the semaphore place is consumed by the *switch* transition, and a token is returned to the semaphore place by the occurrence of the *finish* transition.

The semaphore place ensures that once the *switch* transition has occurred, then no input border transition or the *switch* can occur until after the *finish* transition has occurred. Thus conditions (b) and (c) of Definition 7.31 are trivially satisfied. The implementation can then check Definition 7.31 (a). That is, it can check if once the *switch* transition has occurred leading to M_{sw} that the *finish* transition is live for all markings internally reachable in the supertransition from M_{sw}. If this holds then the *finish* transition is locally live, otherwise it is not.

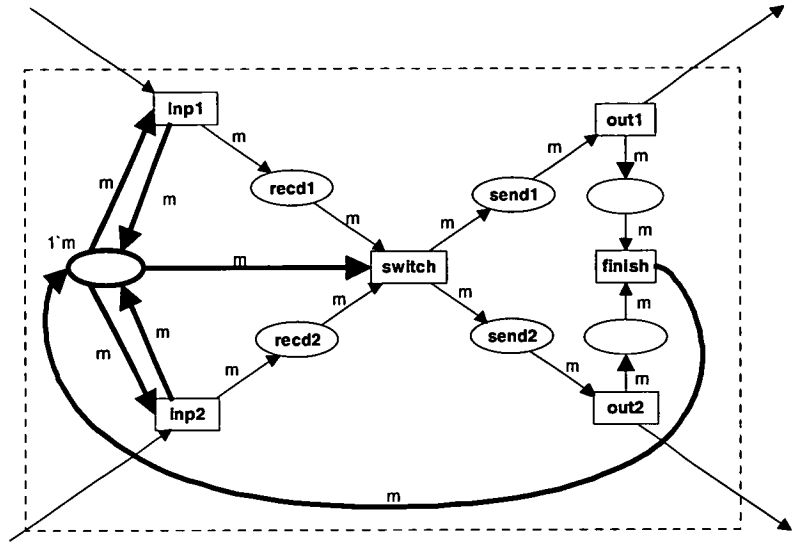


Figure 7.6: Semaphore place added to the canonical basis of a supertransition

7.4.16 Comparison to Modular Analysis

Christensen and Petrucci have developed a reachability analysis technique for analysing modular petri nets. This technique, referred to as Modular Analysis, has been described in Section 6.3.5. Refined nodes can be considered as special types of modules. Our refined node analysis is in many ways similar to Modular Analysis, but there are some important differences between the two techniques. In this section we compare our approach with Modular Analysis. We first examine the difference between the Global Graph of the RNSS and the Synchronisation Graph of Modular Analysis. We then consider the problem Modular Analysis has with place fusion, and why this problem does not appear in refined node analysis. Finally we discuss the differences in how successors are found and the resulting differences in edge labelling between Modular Analysis and our approach.

The Global Graph versus the Synchronisation Graph

Under Modular Analysis, all components are part of a module, and the modules are joined by transition fusion. The synchronisation graph captures the synchronisation between the modules. It contains only occurrences of fused transitions. In contrast to this, we have a global net which can contain modules (i.e. supernodes) and also net components that are not part of a module. One approach to analysing such a net would be to group the components that are not part of a module (i.e the external places and transitions) into a module and then use Modular Analysis. Instead, we develop a global graph which can contain occurrences of any external or terminal transition.

The global graph of the RNSS is similar to the synchronisation graph of Modular Analysis. It provides an overview of the behaviour of the model. This overview will be particularly useful in debugging and analysing the behaviour of a model. In fact, it may be all the developer requires. A possible disadvantage of such a global graph compared to grouping the external components of the global net in another module is that each state of global graph must store the SCC of each supernode, meaning that the global graph may use more memory than is necessary. However, we do not expect this to be a significant overhead in advanced state storage mechanisms. For example, the storage used in

Design/CPN [105] will only store the state of a place if it has changed (otherwise it will effectively store a pointer to the state).

Place Fusion

Modular Analysis has been developed for modular nets. As discussed in Section 2.2.3 modular nets consist of a set of interacting modules. Typically modules interact by shared places (fused places) and/or shared transitions (fused transitions). The modules supported under Modular Analysis are quite general, but must be built with transition fusion. Building modules with place fusion causes problems since even if each of the module graphs have a finite number of states, the full reachability graph may have an infinite number of states. As demonstrated in the net of Figure 7.7 (where the place p_2 is shared), this is because one module can provide enough tokens in a fused place to allow transitions in another module to be enabled, and then this second module can provide more tokens for the first one, and so on.

To allow modules with place fusion to be used in practice Christensen and Petrucci therefore propose a transformation from a modular net using place fusion to a modular net using transition fusion. Figure 7.8 shows the net of Figure 7.7 transformed to use transition fusion.

On the other hand, our modules (i.e. supernodes) arise from node refinement. A superplace can be considered as joined to its environment by place fusion. The constrained behaviour of superplaces will then mean that the environment cannot extract any tokens which it has not deposited or which were not present initially. Thus, the combination of such modules by place fusion ceases to be a problem.

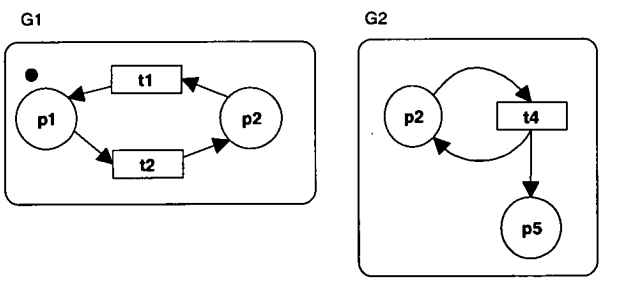


Figure 7.7: Two modules both with finite graphs, but the full reachability graph is infinite [51, p.229]

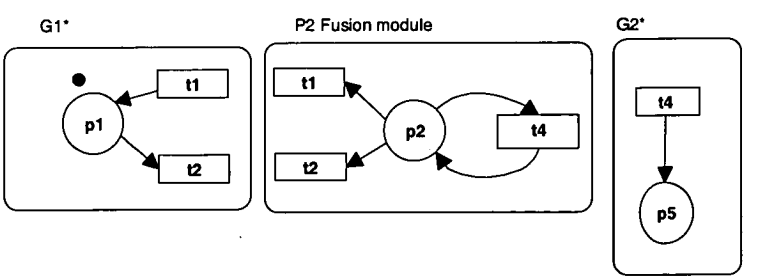


Figure 7.8: The net of Figure 7.7 transformed to use transition fusion [51, p.229]

Successors and Edge Labelling

In the Modular Analysis algorithm, given the marking of all modules, the successors are "the set of all markings reachable by a sequence of internal transitions (or none) followed by a fused transition" [51, p. 231], where internal transitions are those transitions internal to a module and not fused with transitions of other modules [51, p. 231]. This means the successors are those reachable by the occurrence of a sequence of transitions internal to *any* module followed by *any* fused transition.

The Christensen and Petrucci definition could mean that the size of the modular state space is much larger than is necessary. For example, consider the net of Figure 7.9. This net consists of three modules — module A, module B and module C. Modules B and C are fused by the transition *tf*. The modular state space of the net of Figure 7.9 is shown in Figure 7.10. It can be seen that from the initial vertex of the synchronisation graph there is a successor marking and corresponding edge for every internally reachable marking of module A. Clearly, if module A has many internal successors then the number of successors in the synchronisation graph will also be large.

Christensen and Petrucci note this problem in [51, Section 9.2], and suggest is can be avoided with the introduction of a special symbol to denote that a module does not participate in the communication. However, it is not clear how the modular state space algorithm and proofs would be adjusted to cater for this special symbol.

We observe that the state of module A does not affect the enabling of the fused transition *tf*, nor is it affected by the occurrence of *tf*. Therefore it is not necessary to store all these successor markings and corresponding arcs. Instead we need only consider the occurrence of *tf* from markings reachable by a sequence of internal transitions of those modules that have a place that is input to *tf*. By doing this, in the example of Figure 7.9, the only successor of the initial vertex of the synchronisation graph would be (A_1, B_2, C_2). This vertex represents all the markings internally reachable from it (i.e. it represents all successors that appear in the synchronisation graph of Figure 7.10). The edge from the initial vertex to (A_1, B_2, C_2) must also represent all edges that appear in the synchronisation graph of Figure 7.10. This can be achieved by only storing the marking of modules connected to *tf* in the source and successor markings that are stored with the edge, rather than the marking of all modules as is the case in the synchronisation graph of modular analysis.

We adopt this approach of considering the occurrence of a fused transition from markings internally reachable to modules connected to that transition. (For the RNSS we consider the occurrence of a transition connected to superplaces only from markings internally reachable in those superplaces.) This approach can result in a significantly smaller state space. We note that because the source marking stored with each edge of the global graph only stores the marking of superplaces connected to the transition of the edge, then when determining some of the dynamic properties without unfolding the RNSS it is sometimes necessary to determine all possible source markings by considering the markings internally reachable from the source global vertex of the edge. This is in contrast to the Modular Analysis approach where the marking stored with the edge can be examined directly (without considering the internally reachable markings). However, in the Modular Analysis approach there are more edges to consider.

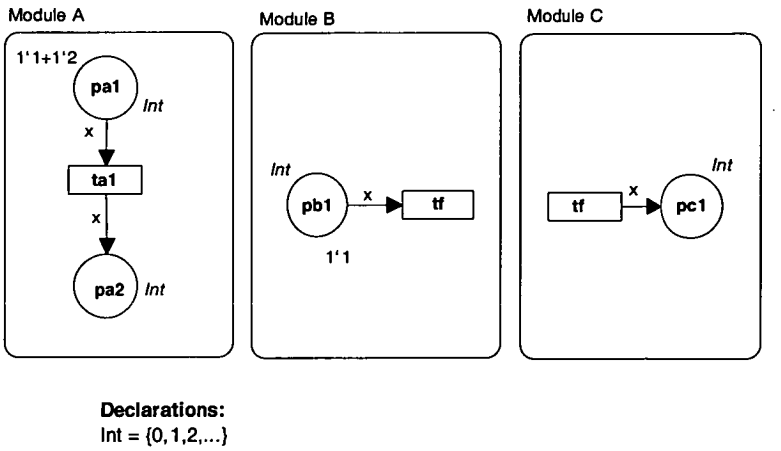


Figure 7.9: A modular CPN

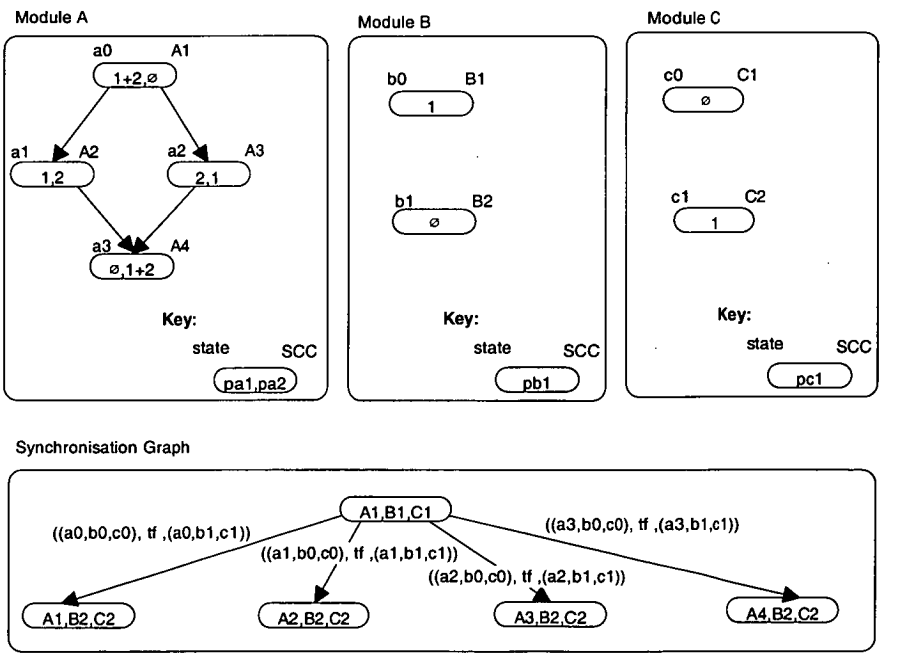


Figure 7.10: The modular state space of the net of Figure 7.9

7.5 Incremental Algorithm

Type, subnet, and node refinement will commonly be used in combination, and it is therefore important that we can combine the previous algorithms to cater for any combination of these refinements. In this section we first indicate how the algorithms that cater for type and subnet refinement can be combined to create a new algorithm that caters for both type and subnet refinement (Section 7.5.1). We then describe an algorithm that caters for all three forms of refinement (Section 7.5.2).

7.5.1 Combining the Type and Subnet Algorithms

In the previous sections we have presented modified versions of the EDGESFROM function to cater for type refinement (Algorithm 7.2) and subnet refinement (Algorithm 7.3). We now combine these two functions to give an algorithm that caters for both type and subnet refinement. We refer to this algorithm as the *type-subnet algorithm.*

To determine the successor markings from a given marking, M, in the refined net, the function that caters for type refinement (EDGESFROM-TYPE) considers the enabled abstract firing elements at the corresponding abstract marking, $\phi(M)$. If the transition of the firing element has not changed by type refinement then the successor marking can be calculated by updating the refined source marking with values from the abstract successor marking, otherwise all refined firing elements that map to each enabled abstract firing element are examined for enabling and the successors calculated in the usual way. Thus the number of refined firing elements examined is constrained by the knowledge of the enabled abstract firing elements. The function that caters for subnet refinement (EDGESFROM-SUBNET) is the same, but it checks for changes due to subnet refinement rather than type refinement, and additionally examines firing elements that do not map to an abstract firing element (i.e. firing elements involving transitions added by subnet refinement).

Since the modified functions have the same basic structure it is not hard to combine them into one function that caters for both type and subnet refinement, as in Algorithm 7.8. As with the EDGESFROM-TYPE function and the EDGESFROM-SUBNET function, the function that caters for a combination of type and subnet refinement (i.e. the EDGESFROM-TYPESUBNET function) considers the abstract firing elements enabled at the corresponding abstract marking, $\phi(M)$. If the transition of the firing element has not changed by type or subnet refinement then the successor marking can be calculated by updating the refined source marking with values from the abstract successor marking, otherwise all refined firing elements that map to each enabled abstract firing element are examined for enabling and the successors calculated in the usual way. As with the function that caters for subnet refinement, this combined function additionally examines firing elements that do not map to a corresponding abstract element. The CHANGED-TYPESUBNET(N,N',t) function returns true if and only if places connected to t have been refined by type refinement, extended by subnet refinement, or if the transition itself has been refined by type or subnet refinement. The function UPDATE(N,N',M_1,M_2',t) is as discussed in Section 7.2.

Algorithm 7.8 Combining the previous EDGESFROM-TYPE and EDGESFROM-SUBNET functions

```
EDGESFROM-TYPESUBNET(N,N',M1,possible)
begin
  Result := ∅
  for all (φ(M1),(t,c'),M2') ∈ ABSTRACTEDGESFROM(N',φ(M1),φ(possible)) do
    if not CHANGED-TYPESUBNET(N,N',t) then
      M2 := UPDATE(N,N',M1,M2',t)
      Result := Result + {(M1,(t,c'),M2)}
    else
      for all (t,c) ∈ FE | φ((t,c)) = (t,c') do
        if M1 ≥ E⁻((t,c)) then
          M2 := M1 − E⁻((t,c)) + E⁺((t,c))
          Result := Result + {(M1,(t,c),M2)}
        end if
      end for
    end if
  end for
  for all (t,c) ∈ FE | (t ∈ possible) ∧ not MAPPED(t,φ) do
    if M1 ≥ E⁻((t,c)) then
      M2 := M1 − E⁻((t,c)) + E⁺((t,c))
      Result := Result + {(M1,(t,c),M2)}
    end if
  end for
  return Result
end
```

7.5.2 Combining the Type, Subnet, and Node Algorithms

We now present an algorithm that caters for type, subnet, and node refinement. First recall that for a net with refined nodes (i.e. supernodes) the state space for each supernode is developed separately, since it is an independent subsystem apart from those points where it interacts with its environment. This leads to several reachability graphs which combine to represent the complete state space of the refined net. We refer to the collection of graphs as the *Refined-Node State Space* (RNSS). The RNSS is composed of a *supernode graph* for each supernode, and a *global graph.* Each supernode graph only contains local information, namely the reachable markings of the supernode and the associated enabled firing elements. Each vertex of the global graph refers to strongly connected components (SCCs) of the supernode graphs, rather than the individual markings. We call such vertices *global vertices.*

An algorithm to construct the RNSS (Algorithm 7.5) was presented in Section 7.4.8. The GLOBALVERTEX function of this algorithm calculates the global vertex corresponding to a given marking. The EDGESFROM-NODE function of this algorithm finds the edges of the global graph from a given global vertex. It first considers each external transition, and then each supertransition.

Since node refinement is also a system morphism then, as with the algorithm that caters for type and subnet refinement (see Section 7.5.1), we can use the knowledge of the abstract firing elements enabled at the corresponding abstract marking to constrain the number of refined firing elements considered. Algorithm 7.9 presents the EDGESFROM-TYPESUBNETNODE function that can be used to find the edges of the global graph when constructing the RNSS. This function uses the abstract graph to constrain the number of firing elements considered.

The EDGESFROM-TYPESUBNETNODE$(N, N', \mathcal{G}_{X''}, v, possible)$ function of returns the global edges and global successors from the global vertex v for any external or terminal transition in the set $possible \subseteq ETT$. For this function to return all the global edges from v, the set *possible* must include all external and terminal transitions that are enabled at a marking internally reachable from v. The CHANGED-TYPESUBNETNODE(N, N', t) function returns true if and only if places neighbouring t have been refined by type refinement, extended by subnet refinement, or form part of a refined node, or if the transition itself has been refined by type or subnet refinement. All other functions of Algorithm 7.9 have been previously presented: INPUTSUPERPLACES is as described in Definition 7.12; FINDANINVERSE is as presented in Algorithm 7.4; EDGESFROM-NODE is as presented in Algorithm 7.7; and ABSTRACTEDGESFROM, UPDATE, and MAPPED are as described for Algorithm 7.8.

First Algorithm 7.9 considers the external transitions t for which there is an edge from the corresponding abstract marking in the abstract graph. If the external transition does not have a superplace as input then the algorithm is similar to that of the EDGESFROM-TYPESUBNET algorithm (Algorithm 7.8), with the exception that the edges store the source and successor marking restricted to $^\circ t^\circ$. (Note that if the transition is not changed by type, subnet, or node refinement then it has no superplace input to it or output from it. Hence $^\circ t^\circ$ is empty and so the marking restricted to $^\circ t^\circ$ is empty.) If the transition does have input from a superplace, then it is added to the set *transToConsider*. These are the external transitions whose enabling must be considered from markings internally reachable from the global vertex v.

After having considered all external transitions for which there is an edge from the corresponding abstract marking in the abstract graph, Algorithm 7.9 considers those transitions t that are added by subnet refinement (i.e. those transitions that are not mapped by the morphism). The definition of subnet refinement (Definition 4.11) will not allow a transition added by subnet refinement to have input from or output to a superplace (if it does then Definition 4.11 (d) is not satisfied). Therefore the global successors due to transitions added by subnet refinement is as for the EDGESFROM-TYPESUBNET algorithm (Algorithm 7.8), with the exception that the edges indicate that the source and successor marking stored with the global edge is empty (since there is no superplace input to, or output from, t).

Finally Algorithm 7.9 adds edges for those transitions in *transToConsider* together with the possibly enabled terminal transitions. This is achieved using the EDGESFROM-NODE function as presented in Algorithm 7.7. Note that a transition is only added to the set *transToConsider* if its corresponding abstract transition is enabled at the corresponding abstract marking. This means Algorithm 7.9 does not consider the internally reachable markings from v for the external transitions that are not abstract enabled.

Algorithm 7.9 EDGESFROM modified for type, subnet, and node refinement

EDGESFROM-TYPESUBNETNODE$(N, N', v, possible)$
begin
  $Result := \emptyset$
  $M :=$ FINDANINVERSE(v)
  $transToConsider := \emptyset$
  for all $t \in (ET \cap possible)$ **do**
    for all $(\phi(M), (t,c'), M_2') \in$ ABSTRACTEDGESFROM$(N', \phi(M), \phi(t))$ **do**
      if INPUTSUPERPLACES$(t) = \emptyset$ **then**
        if not CHANGED-TYPESUBNETNODE(N, N', t) **then**
          $M_2 :=$ UPDATE(N, N', M, M_2', t)
          $Result := Result + \{(M^t, (\emptyset, (t,c), \emptyset), M_2{}^t)\}$
        else
          for all $(t,c) \in FE \mid \phi((t,c)) = (t,c')$ **do**
            if $M \geq E^-((t,c))$ **then**
              $M_2 := M - E^-((t,c)) + E^+((t,c))$
              $Result := Result + \{(M^t, (M|_{\circ t}{}^{\circ}, (t,c), M_2|_{t^{\circ}}{}^{\circ}), M_2{}^t)\}$
            end if
          end for
        end if
      else
        $transToConsider := transToConsider \cup \{t\}$
      end if
    end for
  end for
  for all $(t,c) \in FE \mid (t \in ET \cap possible) \wedge$ not MAPPED(t, ϕ) **do**
    if $M \geq E^-((t,c))$ **then**
      $M_2 := M - E^-((t,c)) + E^+((t,c))$
      $Result := Result + \{(M^t, (\emptyset, (t,c), \emptyset), M_2{}^t)\}$
    end if
  end for
  $Result := Result +$ EDGESFROM-NODE$(N, N', \mathcal{G}_{X''}, M^t, (transToConsider \cup (TT \cap possible)))$
  return *Result*
end

7.6 Summary

In this chapter we have defined the full reachability graph and presented an algorithm to construct it. We then gave algorithms that cater for type refinement, subnet refinement, and node refinement. The algorithms presented in this chapter that cater for type and subnet refinement produce the full reachability graph. We also envisage that those components added by subnet refinement could be grouped in a module and Modular Analysis used to develop the state space. In this case, the full reachability graph would not be produced. The algorithm that caters for node refinement (i.e. the RNSS algorithm) uses a variation of Modular Analysis to produce a set of directed graphs, which we refer to as the *Refined Node State Space* (RNSS). We have proved that the RNSS can be unfolded to give the full reachability graph, and we have also given various propositions, proofs, and algorithms that show how the standard dynamic properties can be determined from the RNSS without unfolding. We presented the RNSS algorithm, and gave an optimisation for it which we proved will find the same dead markings as in the full reachability graph. We compared the RNSS algorithm to the Modular Analysis algorithm and finally we have indicated how the algorithms that cater for type, node, and subnet refinement could be combined.

Chapter 8

Implementing the Incremental Algorithms

Our criteria for a tool to implement the incremental algorithms of Chapter 7 included that it use a CPN or similar formalism, and that it could be easily modified to allow:

- a net to be enhanced by replacing the type of some components with subtypes (in the sense of type refinement)
- a net to be enhanced by adding new components and replacing the type of some components with extended types (in the sense of subnet refinement)
- superplaces and supertransitions to be described
- multiple reachability graphs to be generated, such that several graphs can be referred to while generating another graph
- a marking to be able to be restricted to a given place, or subset of places.

The analysers considered for implementing the algorithms were selected using electronically published surveys [188, 59, 76]. Three analysers were considered in detail: Design/CPN [105], INA [166], and Maria [136]. From the respective tool documentation and detailed discussions with the authors of these tools we came to the conclusion that the Maria tool best satisfied the above criteria since, as explained in the following section, it has a modular design, a simple text based input language, and is implemented in an object oriented language (C++). We therefore chose to implement the incremental algorithms in Maria.

As Mäkelä points out, many issues that appear to be simple or trivial from a purely theoretical point of view can actually require most of the attention when developing software [137]. In this chapter we consider issues in the implementation of the incremental algorithms that are either important from a performance point of view or where we feel others could benefit from a more detailed discussion.

The Maria tool consists of several interacting modules. In Section 8.1 we introduce the Maria analyser by explaining the role of each of the modules. Those modules that were modified while implementing the incremental algorithms are introduced in more detail. In Sections 8.2 – 8.7we examine the modifications to these modules.

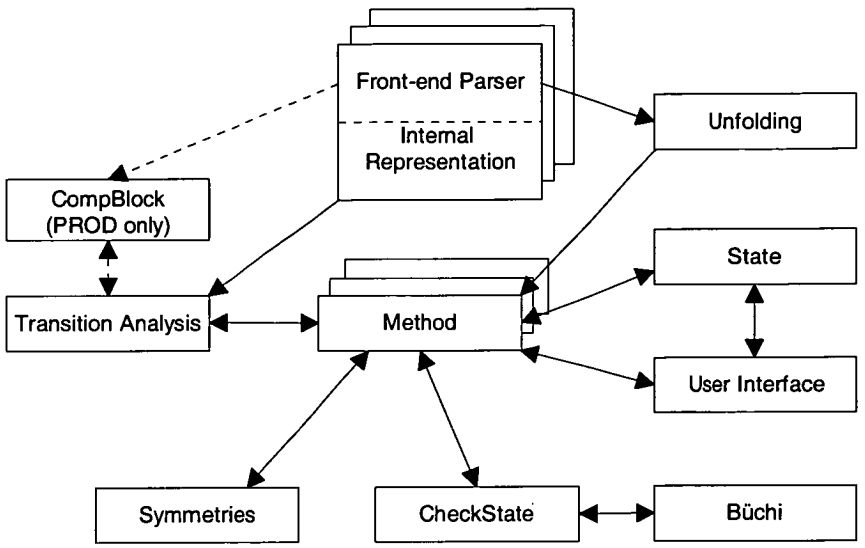


Figure 8.1: The modular structure of the Maria analyser (modified from [96, p. 5])

8.1 The Maria Analyser

The basic function of the Maria analyser is to generate the state space of a HLPN. Questions about the state space can be answered both during the state space generation (on-the-fly) and once the generation is complete. The Maria reachability analyser is a relatively new reachability tool building on the earlier work with PROD [197]. At the time work on the incremental algorithms began, even version 0.1 of Maria was not finished. At the current time, the incremental algorithms are implemented in Maria version 0.1 and all future discussion of Maria in this thesis will be based on this version.

The Maria analyser is implemented in C++ [5], and the main development platform is Linux [187]. The implementation makes extensive use of the C++ Standard Template Library. The Maria parser is constructed using the Flex lexical analyser [154] and the Bison [58] parser generator. For implementing the incremental algorithms we used the GNU C++ compiler [180] as the development environment. For debugging we used the graphical interface to the GNU debugger, DDD [135], and the malloc debugger Electric Fence [156]. In reachability graph generation, even the smallest memory leak would be disastrous, so we used the Insure++ C++ compiler [152] and the DMalloc library [198] to find memory leaks.

One of the most attractive features of the Maria analyser for our purposes was that it has a modular design, with the intention that different algorithms, front-ends, state storage mechanisms, etc, be able to be easily incorporated. Other attractive features include that it has a simple text based input language which would allow the various refinements to be easily identified, and that it has an object-oriented design and implementation, meaning that multiple nets and multiple reachability graphs could be easily supported.

Figure 8.1 shows the modules of the Maria analyser. In the remainder of this section, we first list the purpose of each module, and then describe in more detail those modules that were modified for implementing the incremental algorithms. The actual modifications to each of the modules are described in Sections 8.2 – 8.7.

Front-end Parser and Internal Representation module parses a net description and produces an internal representation of the net.

Method module describes how the analysis is performed. That is, this module contains the implementation of the reachability graph algorithm.

State module stores the state space, both in system memory and on disk. The storage method used is critical. It must be able to handle millions of states and check if a certain state has been previously stored.

Transition Analysis module determines the enabled firing elements at each state.

User Interface module allows the user to examine the reachability graph and check temporal properties.

Unfolding module unfolds the high level net description to a low level net, or partially unfolds the high level net description.

Symmetries module supports the reachability graph reduction using *symmetries* (see Section 6.3.2).

CheckState module implements model checking of the reachability graph, that is, whether the model satisfies a given property. The property is usually given in some temporal logic, such as *Linear Temporal Logic* (LTL).

Büchi supports Büchi automata [39]. LTL model checking is often performed by transforming the LTL formula to an automaton, and so the *Büchi* module interacts with the *CheckState* module.

CompBlock module supports blocks of C code within the input net description. This module is included in Maria for compatibility with the PROD analyser.

8.1.1 The *Front-end Parser* Module

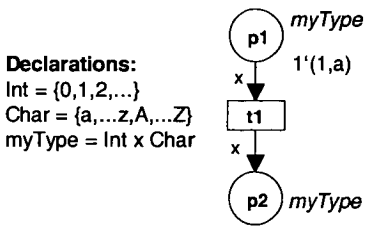
The *Front-end Parser Module* takes a net description and generates an internal representation. In this section we consider the parser and the associated net description language supported by Maria. In the next section (Section 8.1.2) we consider the internal representation of the net.

A front-end parser has been developed for use with Maria. The net is described using a text-only language. The formalism supported by this net description language is a restricted version of Algebraic System Nets [30, 108, 109, 164, 3], but for our purposes it can be considered to be Coloured Petri Nets. The colour sets of a particular net can be defined using data types. The Maria language provides the predefined types `bool`, `int`, `unsigned`, and `char`. All other types are constructed using these types — structures, unions, enumerated types, and buffers can be defined, and constraints imposed. All possible colour sets have a finite domain and are ordered[1].

Figure 8.2 shows a simple net and the associated Maria net description. We will return to this example later in this chapter when describing how the various refinements are supported in Maria. The first line of the net description declares a new type, called `myType`, which is a structure that contains an integer and a character. Following this, two places,

[1]This ordering allows the implementation to easily iterate through the values of the colour. If the size of a colour set is n, then each value can be represented using $\lceil log_2 n \rceil$ bits.

p1 and p2 are declared with token type myType, and the place p1 given an initial marking of one token with the integer value set to 1 and the character value set to *a*. Finally, the transition t1 is declared. This transition transfers a token from place p1 to place p2. (Note that variables are not explicitly typed in Maria, hence the variable *x* is not typed my myType in the net description.)



Maria Net Description

```
typedef struct{int i, char c} myType;
place p1 myType: {1,'a'};
place p2 myType;

trans t1
in {p1: x}
out{p2: x};
```

Figure 8.2: A simple net and its Maria net description

8.1.2 The *Internal Representation*

A class diagram for the classes Maria uses to represent a net is given in Figure 8.3. (Figure 8.3 uses the UML notation, which is described in Appendix A.) Only the main classes have been shown. For example, there is a subclass of the *Type* class for each predefined type supported by Maria, but these subclasses are not shown.

An instance of the *Net* class represents the parsed net. This instance contains a list of places of the net (i.e. a list of references to instances of the *Place* class), as well as a list of transitions (i.e. a list of references to instances of the *Transition* class). Each transition has arcs (instances of the *Arc* class) associated with it. The place (transition) instances have a unique index which identifies the place (transition). Each *Net* instance can have associated with it instances of the *GlobalMarking* class. Each *GlobalMarking* instance represents a marking of the net. Each *GlobalMarking* instance contains a reference to an instance of the *PlaceMarking* class for every place of the net. Each *PlaceMarking* is associated with a given place and stores a marking for that place. (Note that a *GlobalMarking* is simply a marking of the net, and is not related to the global graph of the RNSS described in Chapter 7.) The *Valuation* class stores a possible valuation (firing mode) for a given transition, that is, it stores a value for each variable of a given transition.

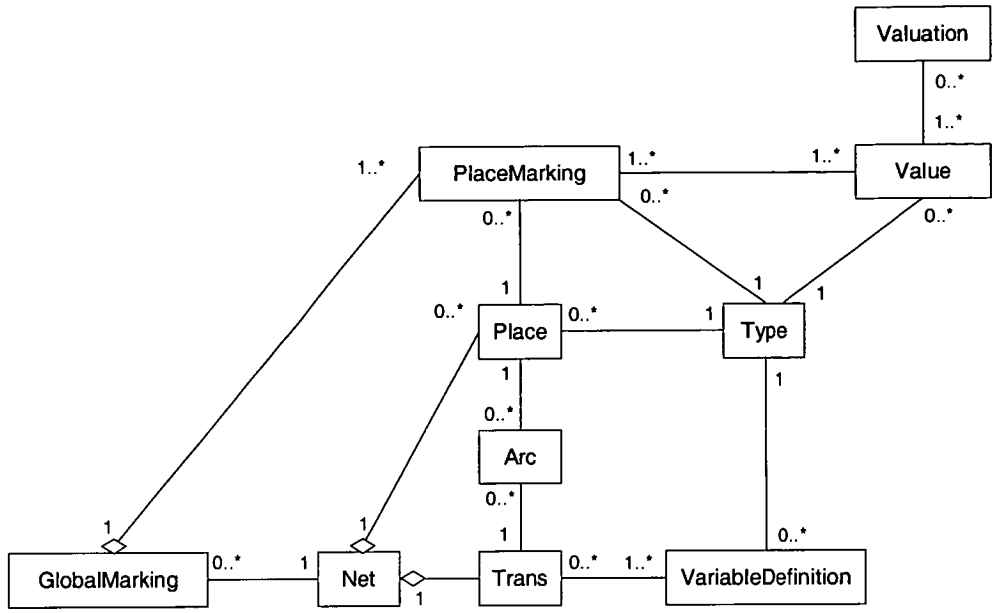


Figure 8.3: The classes Maria uses to represent a net

8.1.3 The *State* Module

The *State* module stores the state space, both in system memory and on disk. The *State* module provides a way of storing the vertices and edges of the reachability graph, determining if a given state is present in the reachability graph, and retrieving the edges and successor states from a given state.

A class diagram for the internal representation used by the *State* module is given in Figure 8.4. There is a one to one relationship between the *Graph* class of Figure 8.4 and the *Net* class of Figure 8.3.

An instance of the *Graph* class represents the reachability graph. The Maria implementation assumes there is only ever one *Graph* instance and all markings and edges are added to this instance. The *Graph* class has an associated hash table (*StateHash*), which stores an instance of the *State* class for each state. The location in the hash table where each state is stored is determined by hashing a bit-string representation of the state.

Since there is an instance of the *State* class for every reachable state, the amount of information stored in the *State* class must be kept to a minimum. The *State* class therefore does not store the actual marking of the state, not even its bit-string representation, but merely records the state number and maintains a list of events (references to instances of

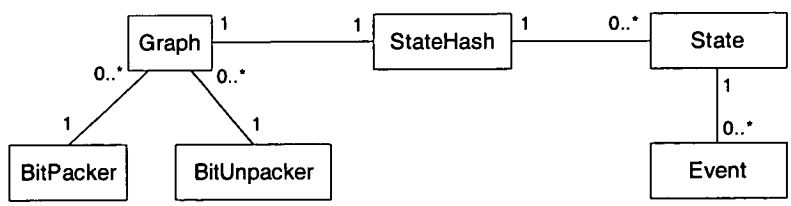


Figure 8.4: The classes Maria uses to store the state space

the *Event* class) that lead to successor states. Each instance of the *Event* class contains a reference to the successor state, but again, since space is at a premium the actual firing element that the event represents is not stored.

Thus the reachability graph structure is stored in system memory, but the actual markings and firing elements represented by the vertices and edges respectively are not. The markings and firing elements are only stored on disk[2]. The *BitPacker* class provides methods for encoding a state (i.e. an instance of the *GlobalMarking* class) to a bit-string. It is this bit-string that is stored on disk. It is also this bit-string that is hashed to determine the location that the state should be stored in the hash table.

The function ENCODE(M,N) presented in Algorithm 8.1 encodes a global marking M of the net N (i.e. it encodes a *GlobalMarking* instance). Encoding a global marking involves encoding each local marking (i.e. *PlaceMarking* instance) associated with the global marking, and concatenating this local encoding to a bit-string. The CONCAT(S_1,S_2) function concatenates string S_2 to string S_1. The order in which places are encoded is not important, provided the same order is used for decoding. (In the implementation, the places are processed in the order that they appear in the list of places that is associated with the net.) The ENCODE(β) function encodes a local marking β where β is a multiset, that is $\beta : A \rightarrow \mathbb{N}$. Encoding a local marking involves first encoding the total number of items, $|\beta|$, to a bit-string, and then appending each distinct item $\{a \mid \beta(a) > 0\}$ and its multiplicities, $\beta(a)$, to this bit-string. The function ENCODEINTEGER(n) returns a bit-string representation of the integer n, and the function ENCODEITEM(a) returns a bit-string representation of the element $a \in A$.

Algorithm 8.1 Encoding a Marking

```
ENCODE(M,N)
begin
  BitString := ""
  for all p ∈ P do
    CONCAT(BitString, ENCODE(M(p)))
  end for
end


ENCODE(β)
begin
  Buf := ENCODEINTEGER(|β|)
  for all {a | β(a) > 0} do
    CONCAT(Buf, ENCODEINTEGER(β(a)))
    CONCAT(Buf, ENCODEITEM(a))
  end for
end
```

[2]One file, given the suffix *.rgs*, stores the vertices of the graph and another, given the suffix *.rga*, stores the edges of the graph. A third file, given the suffix *.rgd*, stores an index to the vertices and edges.

The *BitUnpacker* provides methods for decoding an encoded marking. To decode the marking given a bit-string representation, the order in which the encoded place markings appear in the string (as used in the encoding routine) must be known. For the first place, the total number of items is retrieved, and then each distinct item and its multiplicity is retrieved. By knowing the total number of items, the decoder knows at all times the number of items left to retrieve for that place. When all items have been retrieved, the decoding of the next place can begin.

Similarly, the firing element associated with each edge of the reachability graph is encoded to a bit-string representation and stored in a binary file.

8.1.4 The *Transition Analysis* Module

The *Transition Analysis* module finds the firing elements enabled at each state. This is achieved using the *transition instance analysis* algorithm [136, 137], which determines the enabled firing modes for a given transition. The algorithm is based on a depth-first search. The main idea is to bind tokens in the input places one at a time to the variables on the input arcs of the transition being analysed. The algorithm is complicated since the multiplicity of a given variable can be a variable itself (hence the multiplicity variable must be bound first). In other words, arc expressions can be multiset sums which need to be expanded. It is further complicated by various optimisations.

8.1.5 The *Method* Module

The *Method* module describes how the analysis is to be performed. That is, the *Method* module contains the implementation of the reachability graph algorithm. The algorithm implemented in Maria is essentially the standard reachability graph algorithm (see Algorithm 7.1). In Maria the algorithms of the *Method* module are actually implemented as part of the *Net* class. However, for flexibility, we believe it would be better to separate the implementation of these algorithms from that of the *Net*.

One difference between the algorithm implemented in Maria and Algorithm 7.1 is that there is no EDGESFROM function in the implementation. Rather than determining all the edges possible from a given marking and then adding the edges to the reachability graph, the implementation adds each edge to the reachability graph as soon as it is found.

8.1.6 The *User Interface* Module

The *User Interface* module allows the user to examine the reachability graph and check temporal properties. A program called *Marde* (*Mar*ia *De*bugger) has been implemented for this purpose.

When Marde is invoked it first parses the net that is being examined and then, using the reachability graph files, it re-constructs the internal representation of the state space (see Figure 8.4). Marde provides a query language that allows the user to traverse the internal representation of the reachability graph and to evaluate formulae in the nodes of the graph.

8.2 Modifications to the *Front-end Parser* Module

Having given an overview of the structure of the Maria analyser, we now consider how Maria can be modified to support the following methodology adopted for analysing models developed using type and subnet refinement. First, the abstract and refined nets are parsed. If the incremental algorithm requires the abstract reachability graph then it is developed (if it does not already exist), or the internal representation is recreated in system memory (if it does exist). The refinements from the abstract net are then detected, and the incremental algorithm is used to develop the reachability graph of the refined net.

To support such a methodology the parser had to be modified to allow multiple nets to be parsed (hence creating multiple instances of the *Net* class), and the Maria language was modified to allow easy detection of the various refinements.

The Maria analyser v0.1 assumes that only one net will be parsed and analysed at a time. A single instance of the *Net* class is created to represent the parsed net. All places (transitions) that are parsed are added to the list of places (transitions) associated with this instance. To allow multiple nets to be parsed we modified the parser so that the net instance to which the place and transition instances are added is specified as a parameter.

In the following sections we consider the modifications made to the Maria Language so that type and subnet refinement (Section 8.2.1), and node refinement (Section 8.2.2) can be easily detected.

8.2.1 Detecting Type and Subnet Refinement

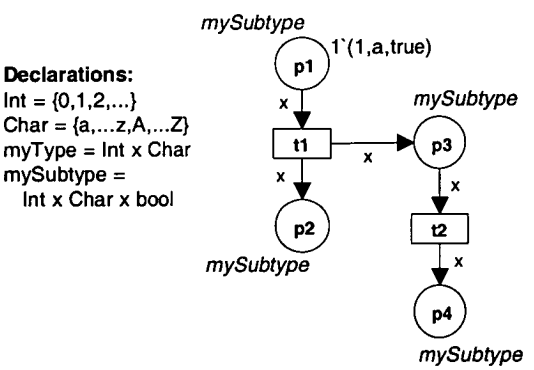
We introduced new syntax to the Maria language to indicate type refinement and subnet extension of existing types. The keyword `subtype` indicates that one type is a type refinement of another, and the keyword `extends` indicates that one type is a subnet extension of another. If a type is defined to be a subtype and/or extension of a given type then the parser sets an attribute of the *Type* instance that indicates the type from which the subtype or extension is derived.

A type and subnet refinement of the net of Figure 8.2 is shown in Figure 8.5, together with the Maria net description. Here the refined net indicates that the type `mySubtype` is a subtype of the type `myType`. The parser will set an attribute of the *Type* instance representing `mySubtype` to indicate that it is derived from `myType`. Note that the places and transitions added and changed by subnet refinement are simply included in the refined net description.

Thus in the refined net description the type of each place indicates whether the place has been refined by subtype refinement, or extended by subnet refinement. The refined *Net* instance is compared to the abstract *Net* instance to determine those components added to the refined net by subnet refinement, and those transitions that have been "changed" in the sense of the incremental algorithms (i.e. those transitions that: are connected to a place that has been refined, have had arcs added, or have had their arc function(s) changed). The initial marking of any places changed by subnet extension is also compared to the initial marking of the corresponding abstract place and any newly added tokens are marked as such. Once the refinements have been detected, the refined net can be checked to determine if the refinements are valid (according to Definitions 4.9 and 4.11).

The detection of new and changed components uses the names of the components to find the corresponding abstract components. Therefore the component names cannot change in the refinement. We have implemented new methods in the *Net* class for finding an instance of a component with a given name, and new equality comparison operators in the *Place*, *Transition*, and *Arc* classes for determining whether two places (transitions) (arcs) are equal. For the newly implemented comparison operators to work correctly, it was necessary to also modify the equality comparison operators of the *Type* classes. These operators originally worked by pointer comparison, and so the instance of a type in the refined *Net* instance would not be considered equal to the same type in the abstract *Net* instance. Therefore we changed these operators to compare the names of the types. Although this string comparison is slower than the original pointer comparison, a production version of the incremental algorithms could overcome this by using the one type instance for the abstract and refined nets.

To make the refinement detection more efficient and easier to implement we assume that the places and transitions of the refined net are listed in the same order as the abstract net, and that any places (transitions) added to the refined net by subnet refinement are listed following the places (transitions) that have corresponding abstract places (transitions). We also require that in cases where a structured type is extended by subnet extension that the last field of the structure is a boolean that indicates whether the token is newly added by subnet refinement. (At the moment the user must specify this field, but it would be easy to automate this in the future.) Clearly these requirements simply make detecting refinements more efficient and simpler to implement. They could easily be removed for a production version of the tool.



Maria Net Description

```
typedef struct{
  int i, char c,} myType;

typedef struct{
  int i, char c, bool b
} mySubtype subtypes myType;

place p1 mySubtype: {1,'a',true};
place p2 mySubtype;
place p3 mySubtype;
place p4 mySubtype;

trans t1
in {p1: x}
out {p2: x;
     p3: x};

trans t2
in {p3: x}
out {p4: x};
```

Figure 8.5: A type and subnet refinement of the net of Figure 8.2 and the Maria implementation

8.2.2 Describing Supernodes

We require supernodes to be explicitly defined by the user in the refined net description. To support this we have added two new constructs to the Maria language — one for superplaces, and one for supertransitions. At the moment it is not possible for the user to declare a type for the supernode, such that the type can be instantiated several times. However, this option can be implemented at a later stage.

A superplace can be described using the syntax:

```
SUPERPLACE name { net };
```

That is, a superplace is defined using the reserved word `superplace` followed by the name of the superplace, and the subnet of the superplace. Similarly, a supertransition can be described using the syntax:

```
SUPERTRANS name { net };
```

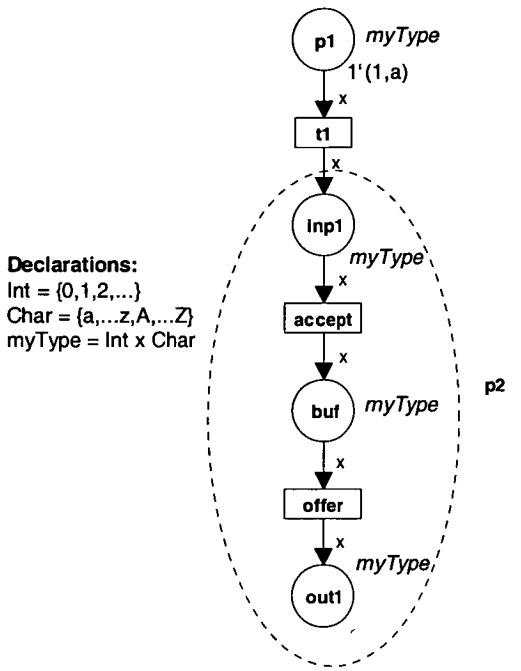
The name of the superplace (supertransition) must coincide with the name of the abstract place which it replaces.

The types and functions defined in the net that contains the supernode (or any containers of this container) are automatically made available in the superplace subnet, and the *dot* ('.') notation is used to allow the components of a supernode to be referenced. For example, `p2.inp1` refers to the place `inp1` in the superplace of `p2`. To allow the border transitions of a supertransition subnet to refer to components in the net that contains the supertransition, we introduce the reserved word `container`. For example, `container.p1` refers to the place `p1` in the container of the supertransition.

When the parser encounters a supernode in the net description it creates a new instance of the *Net* class to represent the supernode subnet. All components of the supernode are added to this *Net* instance. The only requirements the parser enforces on the supernode subnets are that superplaces are place bordered, and supertransitions are transition bordered. However, once parsed, the subnet can be checked to ensure it meets the requirements of canonical supernodes (Definitions 4.13 and 4.14).

Figure 8.6 presents a net with a superplace and the associated Maria net description. The superplace, `p2`, is defined as the net consisting of places `inp1`, `buf`, and `out1`, together with the transitions `accept` and `offer`. Although the type `myType` is declared in the container of the superplace, it is available in the net of the superplace, and the places `inp1`, `buf`, and `out1` are declared to be of this type. As indicated by the description `p2.inp1`, the output arc of the transition `t1` leads to the place `inp1` of the superplace.

Figure 8.7 presents a supertransition and the associated Maria net description. The supertransition, `t1`, is defined as the net consisting of places `recd` and `send` together with the transitions `inp1`, `switch`, and `out1`. Although the type `myType` is declared in the container of the supertransition, it is available in the net of the supertransition, and the places `recd` and `send` are declared to be of this type. As indicated by the description `container.p2`, the output arc of the transition `out1` leads to the place `p2` in the container of the supertransition.



Maria Net Description

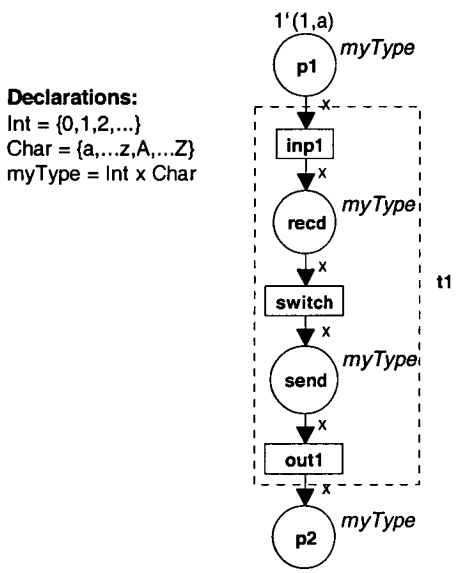
```
typedef struct{int i, char c} myType;
place p1 myType: {1,'a'};
superplace p2{
  place inp1 myType;
  place buf myType;
  place out1 myType;

  trans accept
  in {inp1: x}
  out {buf: x};

  trans offer
  in {buf: x}
  out {out1: x};
};

trans t1
in {p1: x;}
out{p2.inp1: x;};
```

Figure 8.6: A net with a superplace and the Maria net description



Maria Net Description

```
typedef struct{int i, char c} myType;
place p1 myType: {1,'a'};
place p2 myType;
supertrans t1{
  place recd myType;
  place send myType;

  trans inp1
  in {container.p1: x}
  out {recd: x};

  trans switch
  in {recd: x}
  out {send: x};

  trans out1
  in {send: x}
  out {container.p2: x};
};
```

Figure 8.7: A net with a supertransition and the Maria net description

Name Conflicts

The net parser and associated Maria syntax will ensure that no two places (transitions) in the one *Net* instance have the same name. However, since separate *Net* instances are used for supernode subnets, then the name of a component in a supernode can have the same name as a component in the global net or other supernode subnet. To avoid such name conflicts we ensure that each place and transition has a unique name in all supernode subnets and in the global net. To do this, during parsing we prepend the name of the container(s) of that component to its name. For example, if a place *myPlace* is part of a superplace

called *mySuperPlace*, which itself forms part of the global net, *myGlobalNet*, then in the internal representation the name of the superplace will be *myGlobalNet_mySuperPlace*, and the name of the place will be *myGlobalNet_mySuperPlace_myPlace*.

Such a naming convention allows us to uniquely identify each place by name without changing any other code (such as the state storage implementation). It effectively means that each supernode has its own *namespace* (a concept used in C++ [5]). We observe that a similar approach could be adopted to prevent naming conflicts if the Maria language is extended to include modules[3].

8.3 Modifications to the *Internal Representation*

As discussed in Section 8.1.2, Figure 8.3 shows the classes Maria uses to represent a net. In the following subsections we consider the modifications to this internal representation to support incremental analysis. We have added: the *SuperPlace* class to represent superplaces, and the *SuperTransition* class to represent supertransitions (see Sections 8.3.1 and 8.3.2 respectively); and the *GlobalMarkingList* class to store a list of *GlobalMarking* instances (see Section 8.3.3). Figure 8.8 shows the modified Maria class diagram for the internal representation of a net.

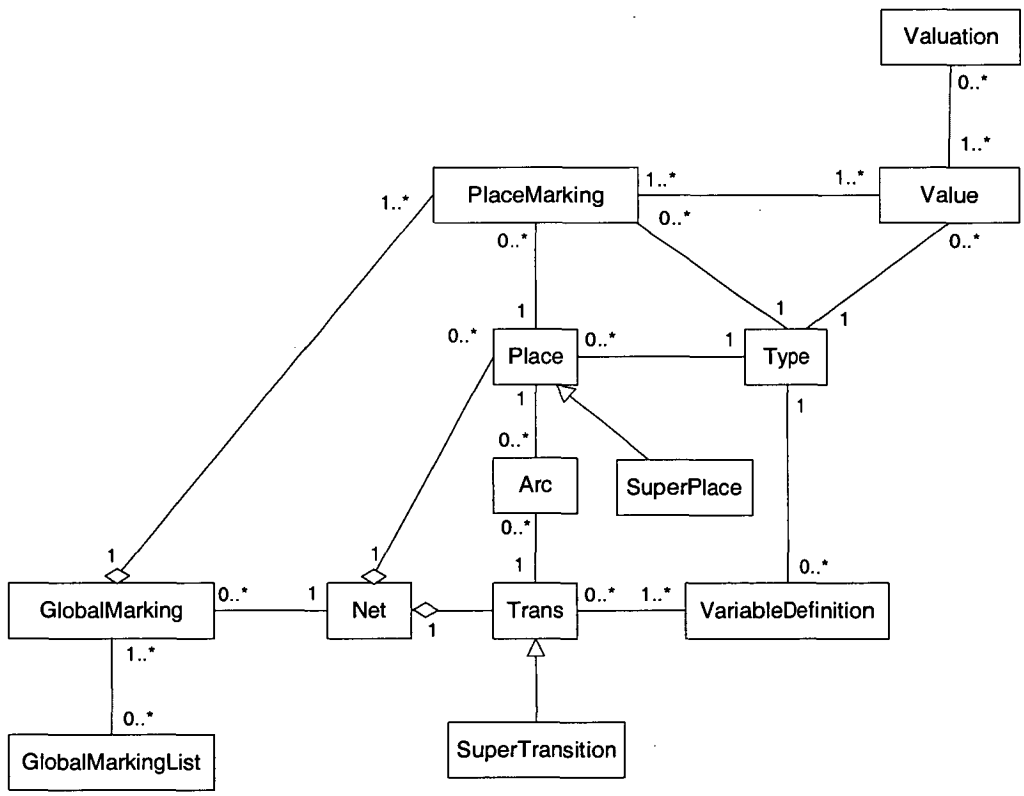


Figure 8.8: The modified class diagram of the classes Maria uses to represent a net

[3]The addition of modules would increase the modelling power of the language, and is highly desired for the translation of Object Petri Nets [113] to Maria nets.

As we explained in Section 8.1.5, the (standard) reachability algorithm implemented in Maria does not use an EDGESFROM function. That is, the implementation does not find all edges from a given marking and then add them to the graph, but instead adds each edge to the graph as it is found. This avoids the overhead of storing all the edges from a given marking and then iterating through these edges to add them to the graph.

The incremental algorithms are specified in terms of modifications to the EDGESFROM function, since this allows us to clearly identify the part of the reachability graph algorithm that is modified by the incremental algorithm. However, as is the case for the implementation of the standard reachability graph algorithm, for the implementation of the incremental reachability graph algorithm there is no need to first find all the edges, and then add them to the graph. Instead the edges can be added to the graph as they are found. Thus in the implementation of the various EDGESFROM functions of the incremental algorithms, each edge is added to the graph as it is found (rather than adding it to a result set).

8.3.1 The Internal Representation of Superplaces

The parser creates an instance of the *SuperPlace* class for each superplace it parses. This class inherits from the *Place* class and therefore has the properties of a place. Additionally, it has the subnet of the superplace as an attribute, and various other attributes and methods for determining properties of the superplace.

Components of a superplace subnet are only added to the *Net* instance that represents the superplace subnet, not to the *Net* instance that represents the global net. This approach allows us to easily manage which components form part of a superplace subnet. It also allows us to recursively call functions on the superplace subnet (e.g. to calculate the superplace reachability graph).

Recall that when a net description is parsed, an instance of the *Place* class is created to represent each place, and a reference to the place instance is added to a list of all places of the net. When a superplace is parsed, an instance of the *SuperPlace* class is created. Since the *SuperPlace* class inherits from the *Place* class, then every instance of a superplace is an instance of a place, and so a reference to the superplace can be added to the list of places of the net. This is useful since a *PlaceMarking* instance for the superplace will be referenced by the *GlobalMarking* instance, and we can use this *PlaceMarking* instance to store the SCC of the superplace. We have also added an attribute to the *PlaceMarking* class to store the subnet marking of the superplace. (This subnet marking is an instance of the *GlobalMarking* class for the subnet of the superplace.) This means that the state encoding and decoding algorithms require minimal changes to handle superplaces (see Section 8.4.3).

8.3.2 The Internal Representation of Supertransitions

An instance of the *SuperTransition* class is created for each supertransition parsed. This class inherits from the *Transition* class and therefore has the properties of a transition. As with superplaces, components of the subnet of a supertransition are not added to the *Net* instance that represents the global net, but instead are added to the *Net* instance that represents the subnet of the supertransition. This *Net* instance is an attribute of the *SuperTransition* class.

Recall that when a net description is parsed, an instance of the *Transition* class is created to represent each transition, and a reference to the transition instance is added to a list of all transitions of the net. When a supertransition is parsed, an instance of the *SuperTransition* class is created. Since the *SuperTransition* class inherits from the *Transition* class, then every instance of a supertransition is an instance of a transition, and so a reference to the supertransition can be added to the list of transitions of the net. The reachability graph algorithm iterates through a list of transitions, determining the enabled firing modes and corresponding successors due to each transition. Since supertransitions can be members of this list, polymorphism can be used so that the occurrence of terminal transitions are found if the list element being considered is a supertransition.

Unlike a normal transition, a supertransition can capture state information. It does this in the subnet that refines the canonical basis. Since a supertransition is not an instance of a *Place* there is no corresponding *PlaceMarking* instance that can be used to store the SCC and subnet marking of the supertransition. We therefore add a place to the global net for each supertransition. This place is purely in the internal representation — the user never knows about it. It stores the marking of the (subnet of the) supertransition, and the SCC index of the supertransition.

A supertransition is bordered by transitions. To be able to develop the supertransition reachability graph, the marking of the environment places is required. Several possibilities for obtaining the marking of the environment places of supertransitions were considered. One possibility was to store a reference to the environment places in both the global *Net* instance and in the supertransition *Net* instance. This solution presented some minor implementation difficulties with memory management. The biggest difficulty of this approach however was that much of the Maria implementation relies on the place index. The place index is an attribute of the place instance that indicates where the place appears in the list of places associated with the *Net* instance. By adding the one place instance to several nets, we require several indices, one for each net in which the place appears. This would increase the complexity of much of the implementation, particularly in the code that handles state storage.

Therefore, to obtain the marking of environment places of a supertransition for constructing its reachability graph, we duplicate each environment place of the supertransition in the supertransition *Net* instance. This solution has the advantage that each newly created place can be assigned its own unique index, and that the already implemented reachability graph function can be used without modification. The main disadvantage is that when the marking of an environment place changes, the marking of the duplicate place in the supertransition subnet does not change, and vice versa. So before the supertransition reachability graph is constructed we set the marking of the duplicate place in the supertransition subnet to be that of its environment place. Following the occurrence of each terminal transition we set the marking of the place in the environment to that of the duplicate place.

8.3.3 Lists of Global Markings

The RNSS algorithm requires all markings internally reachable in a given set of superplaces from a given marking. These internally reachable markings are found as described in Algorithm 7.4 of Chapter 7. Thus we first calculate a list of all those markings internally reachable in one of the superplaces. Then for each marking in the list, all the markings

internally reachable in the next superplace are found. The list is then updated with these newly found markings and the process is repeated for the next superplace, until all superplaces have been considered. We store a list of references to the markings using a newly created class, *GlobalMarkingList*.

Since the one supernode marking may appear in several markings internally reachable from a given marking, we implemented reference counting for *GlobalMarking* instances. Only the reference to the supernode marking (which is a *GlobalMarking* instance itself) is stored in the marking. The number of references to a *GlobalMarking* instance are counted, and the *GlobalMarking* instance is deleted when there are no remaining references to it. This reference counting saves continually cloning the supernode markings.

8.4 Modifications to the *State* Module

As explained in Section 8.1.3, the *State* module implements methods to store the reachability graph of the net. Several changes were made to these methods for the incremental algorithms. The main changes were made so that:

- the marking of a given place could be efficiently changed (Section 8.4.1),
- multiple reachability graphs could be supported (Section 8.4.2),
- the encoding and decoding algorithms handled supernodes (Section 8.4.3),
- SCCs of a reachability graph could be stored (Section 8.4.4).

8.4.1 Replacing the Marking of a Place

As explained in Section 8.1.3, the state storage used in Maria v0.1 encodes the marking of places in a predefined order, and this order is used when decoding the bit-string encoding. This implies that given a bit-string representation of a state, it is not possible to retrieve the marking of a particular place without decoding the marking of all places stored in the bit-string before that place. This is because there is no way to determine where the encoding of a given place starts in the bit-string.

The function that maps from a refined marking to an abstract marking (the morphism ϕ), requires that the marking of certain places (those refined by type and/or subnet refinement) are replaced with their corresponding abstract marking. Those places that are not refined are not changed. Similarly, the UPDATE function of the incremental algorithms requires us to replace in a given marking the marking of certain places while leaving the marking of other places unchanged (see Section 8.5.2). However, due to the state storage of Maria, given an encoded marking to replace the marking of a particular place we are required to first decode the entire marking, then change the marking of the place, and finally encode the entire marking. Clearly this is inefficient and a more sophisticated state storage mechanism is desirable, where the marking of a given place can be retrieved or changed.

We therefore modified the existing marking encoding algorithm (Algorithm 8.1) to allow the marking of a given place to be determined and/or changed from the encoded bit-string without first decoding the marking of other places. This was achieved by prepending a list of offsets to the bit-string, one for each place, indicating where in the bit-string the

encoding of that place begins. This allowed us to implement various functions to directly decode and replace the marking of a given place. (In this case the offsets usually have to be recalculated.)

Including these offsets will increase the amount of disk space required to store the reachability graph and the time taken to compute the reachability graph. However, in tests we have run (e.g. the distributed database manager net presented in the following paragraphs) the extra disk space used and time taken is small and often negligible. Since a hashed value of the bit-string is used to represent a marking in system memory, and the offsets do not need to be used when calculating the hash value, then including these offsets does not affect the amount of system memory used to store the reachability graph. (Often system memory is more critical than disk space.)

A modified version of the distributed database net originally presented by Jensen [102] is provided with the Maria distribution. A graphical representation of modified net is given in Figure 8.9. It models a simple distributed database with n different database managers, each of which has its own local copy of the database. The set $DBM = \{d_1, d_2, \ldots, d_n\}$ is a set of database managers. Each manager is allowed to make an update to its own copy of the database, but then it must send a message to the other managers so they can perform the same update in their local copy of the database. The *Exclusion* place ensures that all other managers update their local copy before another manager a new change. Initially all database managers are *Inactive* and the *exclusion* place contains a null token (indicated by empty parentheses). A manager can then fire the *Update and Send Messages* transition, in which case its state changes to *Waiting*. Now the manager must wait until all other managers have acknowledged the update (*Receive all Acknowledgements*) before it can return to being inactive.

Figure 8.10 shows the amount of extra disk space that is used by storing offsets for a distributed database net. Figure 8.11 shows the extra time required to compute the reachability graph when offsets are used for the same net. (Both sets of results are for the standard reachability graph algorithm.) As one would expect, as the reachability graph grows, the impact of storing offsets becomes more noticeable. The raw data from Maria, together with percentage differences are given in Appendix C, Tables C.1 and C.2.

Finally, we note that it was not essential to make this modification to implement the incremental algorithms, but it did allow the UPDATE and ϕ functions of the incremental algorithms to be implemented efficiently. Since more sophisticated state storage mechanisms (e.g. Design/CPN [105]) often allow the marking of a given place to be efficiently retrieved and changed, and since our tests indicate the impact of storing offsets is relatively small, we feel such a change is justified for testing the performance of the incremental algorithms.

8.4.2 Creating Multiple Reachability Graphs

Maria assumes there is a single reachability graph (i.e. a single instance of the *Graph* class), all markings and edges are added to this instance. However, the RNSS algorithm requires a reachability graph for each supernode. To support this we create an instance of the *Graph* class for every supernode. We modified the algorithm implementation so that the *Graph* instance to which the markings and edges are to be added can be specified as a parameter. We also maintain a list of all *Graph* instances, which allows a particular instance to be quickly and easily found.

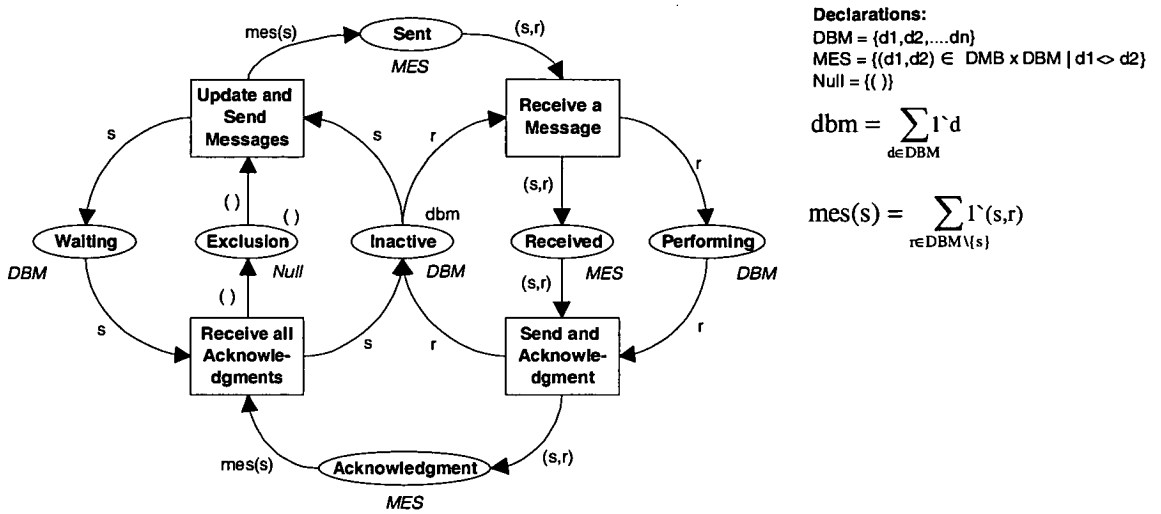


Figure 8.9: Jensen's Database Manager

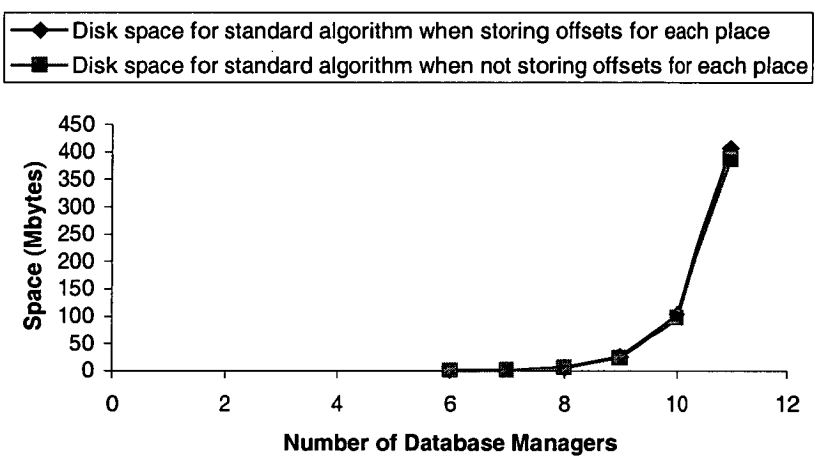


Figure 8.10: Disk space overhead of storing place offsets for each marking of the Database Managers Net

8.4.3 Modifications to the Encoding and Decoding Algorithms

To encode a global marking of a net with supernodes, we use Algorithm 8.1. Since the *PlaceMarking* instance stores the SCC of the superplace marking in its reachability graph, this algorithm encodes the SCCs of superplaces as required. Similarly since the *PlaceMarking* used to store a supertransition stores the SCC of the supertransition marking in the supertransition reachability graph, the SCC index of the supertransition marking is encoded as required. For decoding, when a superplace (or place used to store a supertransition marking) is encountered, the SCC value is decoded and optionally a superplace (or supertransition) marking corresponding to that SCC is also decoded.

The edges of the global graph require the source and successor markings to be stored with the edges. To achieve this we modified the edge encoding algorithms so that the source and successor marking of the net can be appended to the bit-string encoding of the edge. Rather than store the actual source and successor marking with the edge, we save space by simply storing their corresponding state number.

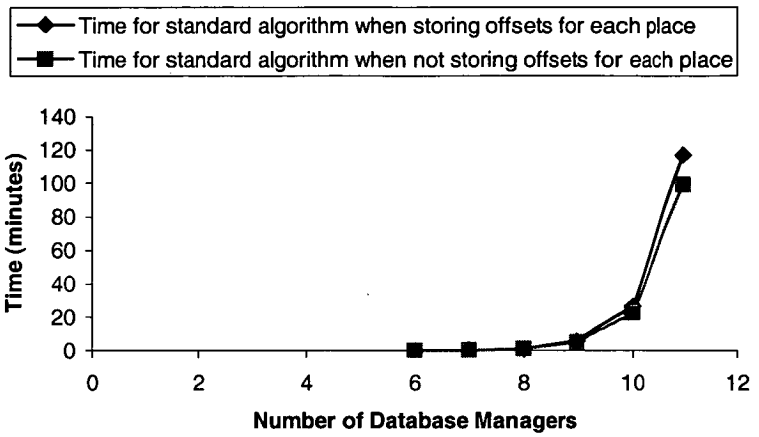


Figure 8.11: Time overhead of storing place offsets for each marking of the Database Managers Net

8.4.4 Storing Strongly Connected Components

We have explained how we store the SCC number of the marking of a supernode in the global marking. Before the SCC number of the marking of a supernode can be stored, the SCCs of each state of the supernode reachability graph must be calculated and stored.

Several options were considered for storing the SCC number for each state of the supernode graph. The option we chose was to prepend the SCC number to the bit-string encoding of the marking. If SCCs are to be calculated we first prepend a dummy value to the encoded state bit-string. Once the SCC for that state is calculated, we overwrite this dummy value with the actual SCC number. The hash function was modified so that this prepended value is ignored. It is also necessary to store in a separate file a map containing one state number for each SCC number. This map allows us to determine a state of a given SCC, which is required to determine an actual marking corresponding to a global vertex (which in turn is needed to find the internally reachable markings from a global vertex as required by the RNSS algorithm). Disadvantages of storing SCCs in this manner are that one must know whether SCCs are to be calculated for the graph before the reachability graph is generated, and that the dummy value is written before the actual value has been found. An advantage is that we do not need a separate file to store a map of all state numbers and their corresponding SCC numbers. In this way we minimise disk usage and disk access.

8.5 Modifications to the *Method* Module

The *Method* module contains the algorithms that calculate the reachability graph. (As we discussed in Section 8.1.5, these algorithms are implemented as part of the *Net* class.) We have implemented algorithms that take advantage of type, subnet, and node refinement, as well as a combination of these refinements. This required implementing the various functions described in Algorithms 7.2 – 7.9, as well as the algorithms themselves. In particular we implemented functions to determine the corresponding abstract marking of a given refined marking, functions to determine the superplaces input and output to a given transition, and the functions: UPDATE, CHANGED-TYPE, CHANGED-SUBNET, CHANGED-TYPESUBNET, CHANGED-TYPESUBNETNODE, MAPPED, COMPUTESCCS,

INPUTSUPERPLACES, GLOBALVERTEX, INTERNALLYREACHABLE, FINDANINVERSE and ABSTRACTEDGESFROM.

The implementation details of the majority of these functions do not warrant discussion here. In Section 8.5.3 we consider the implementation of the CHANGED and MAPPED functions. Sections 8.5.2, 8.5.4 and 8.5.5 examine the implementation of the UPDATE, ABSTRACTEDGESFROM and COMPUTESCCS functions respectively. Firstly we consider finding those refined firing elements that map to enabled abstract firing elements, as required by the algorithms that cater for type and subnet refinement.

8.5.1 Finding Enabled Refined Firing Elements

Recall that Algorithms 7.2 and 7.3 present modifications to the EDGESFROM function of the standard algorithm (Algorithm 7.1) to cater for type and subnet refinement respectively. These functions have also been combined into a single function — the EDGESFROM-TYPESUBNET function — that caters for both type and subnet refinement. The same principles used in the EDGESFROM-TYPESUBNET have been included in the algorithm that caters for node refinement to give an algorithm that caters for type, subnet, and node refinement (Algorithm 7.9).

In this section we discuss how currently the data structures of Maria do not provide optimum support for the incremental algorithms that cater for type and subnet refinement. This affects both the algorithm that caters for type refinement (Algorithm 7.2), and the algorithm that caters for subnet refinement (Algorithm 7.3). It therefore also affects the algorithm that caters for both type and subnet refinement (Algorithms 7.8), and the appropriate part of the algorithm that caters for type, subnet, and node refinement (Algorithm 7.9).

The algorithms that cater for type and subnet refinement allow us to efficiently determine the enabled refined firing elements at M by examining only those refined firing elements that map to an enabled abstract firing element at $M' = \phi(M)$. In order to do this, we need to be able (within the context of markings) to map from refined token elements to abstract token elements and back again.

We need to be able to map from refined token elements to abstract token elements in order to determine M'. We can then lookup M' in the abstract reachability graph to determine the enabled abstract firing elements. We need to be able to map from abstract token elements to refined token elements so that the instance analysis algorithm can be modified as follows: it only considers binding a refined token if the token corresponds to an abstract token that has been bound in an enabled abstract firing element, and if the refined tokens already bound so far also correspond to abstract tokens bound in the same enabled abstract firing element.

While mapping from refined token elements to abstract token elements is easily supported (the data added by type refinement is simply ignored) the reverse direction is not. The current data structures mean that to determine the refined token elements that map to a given abstract token element on-the-fly, we must iterate through all the refined token elements, testing each one. However, this is the same effort as the instance analysis algorithm, since it iterates through all refined tokens to determine if they can be bound. Performance improvements can therefore be expected with better support for these mappings. This is a matter for further research.

However, we are still able to gain some advantage in the implementation from the type and subnet refinement. First and foremost, we do not have to use the transition instance analysis algorithm on transitions for which the corresponding abstract transition (if it exists) is not enabled. (This does not apply to transitions and/or firing modes which are introduced by subnet refinement.) Secondly, if the transition has not been changed then we do not have to check if it is enabled in the refined net, and can obtain the refined successor marking by updating the abstract successor marking.

Here we only present the implemented version of the EDGESFROM-TYPESUBNET function, since this shows how the implementation caters for both type and subnet refinement. The implemented function is given in Algorithm 8.2. This algorithm only considers those transitions that are abstract enabled. If the transition is not changed, then the refined successor is found by updating the abstract successor (using the function UPDATE as presented in Algorithm 8.3). If the transition is changed then the same method as the standard algorithm is used to find the enabled firing elements. This is achieved by calling the EDGESFROM† function, which will result in the transition instance analysis algorithm being used. The EDGESFROM† function is slightly different from that previously presented (see Algorithm 7.1). This is indicated by appending the dagger symbol ($\dagger$) to the function name. The difference is that it takes an additional parameter, *considerAllModes*, which is a set of transitions for which the transition instance analysis algorithm needs to consider every possible mode. The EDGESFROM† function will only consider those firing modes newly added by subnet refinement for any transition not in *considerAllModes*. (Those transitions in the set *considerAllModes* correspond to those transitions which are abstract enabled.)

Algorithm 8.2 The implemented EDGESFROM-TYPESUBNET algorithm

EDGESFROM-TYPESUBNET$(N, N', M, possible)$
begin
  $Result := \emptyset$
  $considerAllModes := \emptyset$
  for all $(\phi(M), (t, c'), M_1') \in$ ABSTRACTEDGESFROM$(N', \phi(M))$ **do**
    if not CHANGED-TYPESUBNET(t) **then**
      $M_1 :=$ UPDATE(N, N', M, M_1')
      $Result := Result + \{(M, (t, c'), M_1)\}$
    else
      $considerAllModes := considerAllModes + \{t\}$
    end if
  end for
  $transToConsider := \{t \in T \mid$ not MAPPED(t, ϕ) $\vee$ CHANGED$(t)\} \cap possible$
  $Result := Result +$ EDGESFROM$^{\dagger}(N, M, transToConsider, considerAllModes)$
  return *Result*
end

8.5.2 Implementing the UPDATE Function

The UPDATE(N, N', M_1, M_2', t) function takes as parameters the refined source marking, M_1, the abstract successor marking M_2', and the transition that led to the abstract successor, t, and returns the refined successor marking. A simple algorithm for the UPDATE function

is the function UPDATE1 of Algorithm 8.3. This function first copies the refined source marking and then replaces the marking of each place neighbouring the transition t with the marking of that place from the abstract successor marking.

A disadvantage of this algorithm is that the refined source marking must first be copied. (We cannot change refined source marking M_1 directly since the refined source marking is required later in the algorithm.) Another possible algorithm for the update function is the function UPDATE2 of Algorithm 8.3. This function updates the abstract successor, M_2', to form the refined successor. It does not need to first copy the abstract successor marking since this marking is not required again. The UPDATE2 function first copies from the refined source, M_1, the marking of each place added to the refined net by subnet refinement, and the marking of each place internal to a superplace. It then removes from M_2' the marking of each abstract place that is refined by node refinement (i.e. each place in P''), and finally the marking in M_2' of each place that is changed by type or subnet refinement is replaced with the marking of that place from the refined source, M_1. (Here the CHANGED-TYPESUBNET(N,N',p) returns true if and only if the type of the place p has been refined by type refinement, or extended by subnet refinement).

We have implemented both update functions of Algorithm 8.3 in Maria, and found that generally the UPDATE2 function performs better than UPDATE1 since UPDATE2 does not copy the refined source marking. For example in the Z39.50 protocol refined with segmentation the type-subnet algorithm (Algorithm 7.8) constructs the refined state space in 587 seconds when UPDATE2 is used, and takes 763 seconds when UPDATE1 is used. (The nets used to model the Z39.50 protocol and the protocol extended for segmentation are presented in detail in the results section, Section 9.3.) The results presented in the next chapter use the UPDATE2 function for updating the refined source marking.

8.5.3 Implementing the CHANGED and MAPPED Functions

Since the CHANGED and MAPPED functions of the incremental algorithms will be called many times during the reachability graph construction, it is important that they are as efficient as possible. As we noted in Section 8.2.1 those transitions that are changed by type or subnet refinement, as well as those newly added transitions (i.e. those that are not mapped to the abstract net), are detected and marked as such once the refined net is parsed. Therefore the various CHANGED functions simply need to check the transition instance to determine if the transition has been changed by type and/or subnet refinement respectively. No extra work is required. Similarly the function MAPPED can simply look at the place or transition instance to determine if it is mapped to an abstract place or transition.

8.5.4 Implementing the ABSTRACTEDGESFROM Function

The are several possibilities for the implementation of the ABSTRACTEDGESFROM function. These are examined in this section.

We believe it is likely that the abstract model will be analysed before the refined model is analysed. If this is the case, then the abstract reachability graph can be used to determine the enabled abstract edges from a given abstract marking. That is, the ABSTRACTEDGESFROM function can simply look up the abstract edges in the abstract

Algorithm 8.3 UPDATE Functions

UPDATE1(N,N',M_1,M_2',t)
begin
  $M_2 := M_1$
  for all $p \in {}^\bullet t \cup t^\bullet$ **do**
    $M_2 := M_2 - M_2|_p + M_2'|_p$
  end for
  return M_2
end

UPDATE2(N,N',M_1,M_2',t)
begin
  for all $p \in P \mid$ notMAPPED$(p,\phi) \vee \phi(p) \in X''$ **do**
    $M_2' := M_2' + M_1|_p$
  end for
  for all $p \in P''$ **do**
    $M_2' := M_2' - M_2'|_p$
  end for
  for all $p \in P \mid$ CHANGED-TYPESUBNET(N,N',p) **do**
    $M_2' := M_2' - M_2'|_p + M_1|_p$
  end for
  return M_2'
end

reachability graph. If, on the other hand, the abstract graph is not already known, then there are two main options:

1. calculate the enabled abstract firing elements (using the instance analysis algorithm on the abstract net) and the corresponding abstract successor markings as required.

2. calculate the abstract reachability graph and then use it to determine the abstract edges.

There are some subtle differences between these two options. The most important difference is that the first method will require a representation of both the abstract graph and the refined graph to be stored in system memory, while the second method will only require a representation of the refined graph. On the other hand, since many refined markings can map to the one abstract marking, the second method may involve calculating the edges and successors for the one abstract marking many times.

The performance of each method will be dependent on the net being analysed. We can create a net where the performance of the first method will be superior. Such a net will have many abstract states that do not have corresponding refined markings, and few refined states of the net will map to the same abstract state. Alternatively, we can create a net where the performance of the second method will be superior. Such a net will have a number of refined markings that map to the one abstract marking and few abstract markings that do not have a corresponding refined marking. In general we believe the second method will have better performance, the reason being that when subnet refinement is used it will often be the case that many refined states map to the one abstract state. This view is supported in

tests we have run on nets refined using subnet refinement, where the incremental algorithm that uses the second method to determine the enabled abstract transitions is significantly faster than the first. For example, in the Z39.50 protocol refined for segmentation, the type-subnet algorithm using the first method takes a total of 2 084 seconds, whereas using the second method it takes 844 seconds. (The nets used to model the Z39.50 protocol and the protocol extended for segmentation are presented in detail in the results section, Section 9.3. This section also compares the performance of the type-subnet algorithm to that of the standard algorithm.)

If the abstract graph is not calculated before the refined net is analysed then we calculate the abstract graph. Again there are two main options. First, we can calculate the abstract graph, and then calculate the refined graph (where the ABSTRACTEDGESFROM function uses the abstract graph to lookup the abstract edges). Alternatively, we can calculate the abstract and refined graphs in concert. That is, the ABSTRACTEDGESFROM function first looks for the abstract marking in the abstract graph. If the marking and its immediate successors are not in the abstract graph, then they are added to the abstract graph. This second option has the advantage that any part of the abstract graph for which there is no corresponding refined behaviour is not calculated. However, since we expect the abstract graph will usually be calculated before the refined graph is developed then the results presented in Chapter 9 use the first option. This allows us to examine how the time taken to develop the abstract graph affects the performance of the incremental algorithms.

8.5.5 Implementing the COMPUTESCCS Function

The RNSS algorithm calls the COMPUTESCCS function to compute the SCCs of a supernode reachability graph. We use Tarjan's algorithm [184] (described in Appendix B) to calculate the SCCs. This algorithm calculates SCCs for a complete graph. However, we require the SCCs for the supernode reachability graph after it is developed for each different abstract marking. To ensure that each SCC is assigned a unique number we store the last used SCC index in the *Graph* instance. This value is then incremented and used when the SCCs are next calculated for that net. We note that the already calculated SCCs will not change when the marking of the supernode is changed due to external input.

8.6 Modifications to the *Transition Analysis* Module

The *Transition Analysis* module uses the instance analysis algorithm to determine the enabled firing elements at a given marking. The only changes made to the instance analysis algorithm were so that we could additionally specify that only tokens newly added in the refinement should be considered (as required by EDGESFROM†, presented in Section 8.5.1). This was achieved by adding an additional parameter to the methods that implement the instance analysis algorithm. If this parameter is true then only modes involving tokens added by subnet refinement are considered when determining the enabled firing modes of a given transition.

8.7 Modifications to the *User Interface* Module

As explained in Section 8.1.6, the user interface program, Marde, allows a reachability graph of a net to be examined. It is assumed that for each net there is only one reachability

graph. However, for the RNSS algorithm each global net can have several subnets (one for each supernode), and a reachability graph is associated with each subnet. We therefore modified the Marde program so that the user can specify that they wish to examine the reachability graph of a particular subnet, or that they wish to examine the global graph.

8.8 Summary

In this chapter we considered the implementation of the incremental algorithms. The Maria reachability analyser [136] was selected as the most appropriate of the available tools for the implementation of the algorithms. The Maria analyser has a modular design, and we have modified the *Front-end Parser and Internal Representation*, *State*, *Method*, *Transition Analysis*, and *User Interface* modules in order to implement the incremental algorithms.

The modifications to the *Front-end Parser* involved adding constructs to the Maria language to support the detection of the various refinements. Classes for superplaces, supertransition, and lists of global markings were added to the internal representation generated. The *State* module was modified to support the efficient implementation of the UPDATE function, multiple reachability graphs and SCCs.

Implementing the incremental algorithms involved implementing several functions. The ABSTRACTEDGESFROM function is implemented by looking up the edges in the abstract reachability graph. SCCs are computed using Tarjan's algorithm. Currently the implementation does not provide support for mapping from abstract token elements to refined token elements (in the context of markings). The algorithms were therefore modified from those presented in Chapter 7. The modifications required that the *Transition Analysis* module be changed so that it can be specified that only firing modes involving newly added tokens should be considered. Finally, the *User Interface* module has been modified so that the reachability graph associated with a supernode can be examined.

In the next chapter we examine the performance of the incremental algorithms.

Chapter 9

Performance of the Incremental Algorithms

In Chapter 7 we presented algorithms that cater for type, subnet, and node refinement, as well as an algorithm that caters for a combination of type and subnet refinement, and one that caters for a combination of all three forms of refinement. Recall that we refer to these algorithms as *incremental algorithms*, and that we refer to the algorithm that caters for a combination of type and subnet refinement (Algorithm 7.8) as *the type-subnet algorithm*, and to the algorithm that caters for node refinement (Algorithm 7.5) as the *RNSS algorithm*.

The aim of developing incremental algorithms is to decrease the time and space required to construct the state space, and therefore help alleviate the state space explosion. In this chapter we examine the performance of the incremental algorithms. We first examine the cost and potential benefit of computing SCCs of the supernode reachability graphs (Section 9.1). Then in Section 9.2 we characterise the circumstances under which performance improvement can be expected and demonstrate the improvements using specially constructed examples. We have implemented two separate case studies to assess the performance of the incremental algorithms in practice: *the Z39.50 Protocol for Information Interchange* [122] and *the Distributed Missile Simulator Model* [82]. Both of these case studies have been developed incrementally (see Chapter 5). The performance of the incremental algorithms for these studies is considered in Sections 9.3 and 9.4 respectively. Some of the results of this chapter have been previously published [129].

The results quoted in this chapter have been obtained using a 500MHz Intel 686 machine running Linux [187] (kernel 2.2.15). The machine has 256 Mbytes of random access memory (RAM), 2 Gbytes of virtual memory (which is the limit for this kernel), and approximately 4 Gbytes of free disk space. The code was compiled using the optimising option of the GNU C++ compiler (g++) [180] (version egcs-2.91.60), and only the essential operating system processes plus the actual test were running on the machine. The time measurements presented are the total elapsed time taken for the algorithm itself to run. We do not include time for things such as parsing the net. The raw data from Maria together with percentage differences for all graphs of this chapter is given in Appendix C.

9.1 Cost of Computing SCCs

Strongly connected components can be used as a way of grouping vertices of a reachability graph so that not as many vertices need to be considered when developing the reachability graph [103]. They can also be used to advantage in determining properties of the reachability graph [103], but for a graph where the size of the SCCs is small, the benefit of computing SCCs is also small. In the worst case, for an acyclic directed graph, all SCCs are trivial and the SCC-graph is isomorphic to the original graph. In such situations we pay the price of computing the SCCs without any gain in performance. The cost of computing SCCs for a standard reachability graph has been discussed in [103]. Here we are interested in the cost of calculating the SCCs of the supernode graphs in the RNSS algorithm.

The benefit of calculating the SCCs of the supernode graphs in the RNSS algorithm is that the nodes of the supernode graph can be grouped according to their SCC. This means that only one vertex is required in the global graph for each SCC of the supernode graph rather than one vertex in the global graph for every vertex in the SCC.

The cost of computing the SCCs will partly depend on the implementation and algorithm used (see Section 8.5.5). In general we expect the cost of computing the SCCs to be relatively low, whereas the potential benefits could be great. Therefore, we have included the calculation of SCCs in the definitions and algorithms. This is confirmed in tests we have performed. For example, the superplace of Figure 9.1 has non-trivial SCCs. As shown in Figs. 9.2 and 9.3, as we increase the number of transitions in the sequence t_1 to t_n (i.e. as we increase the value of n) then we increase the number of states in the SCCs of the superplace graph. This in turn means that as we increase n the time and space performance improves for when SCCs are calculated compared to when they are not.

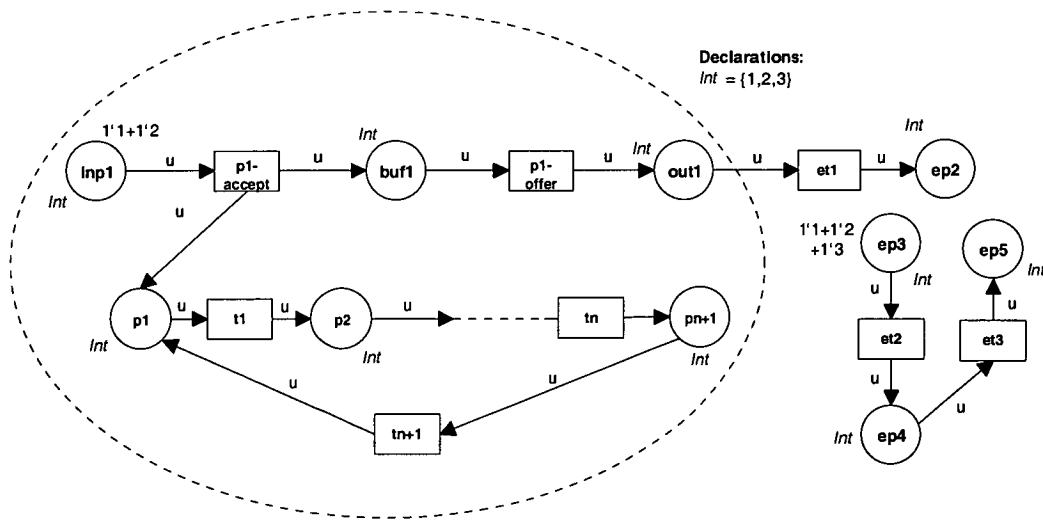


Figure 9.1: A net where it is beneficial to compute SCCs

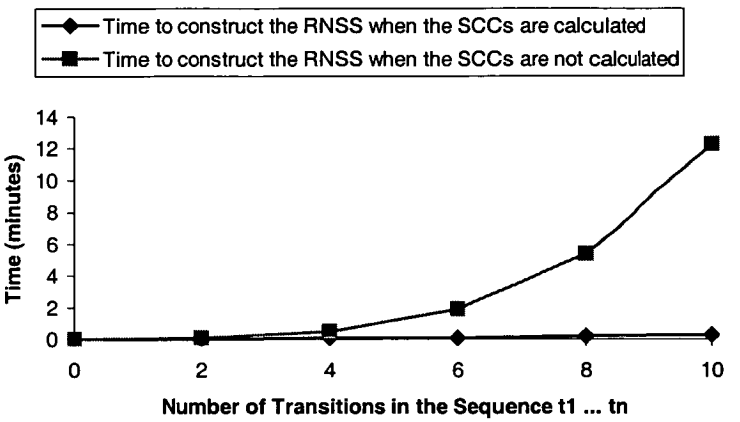


Figure 9.2: Time improvement when computing SCCs for the net of Figure 9.1

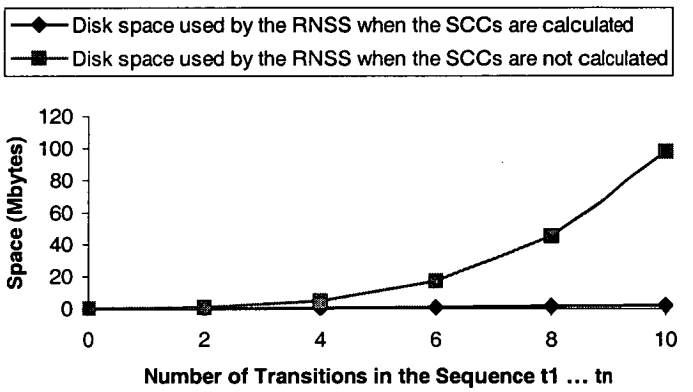


Figure 9.3: Space improvement of computing SCCs for the net of Figure 9.1

However, it is possible to pay an overhead for computing the SCCs for little benefit. For example, consider the net of Figure 9.1, where the transition t_{n+1} has been removed. In this net, the superplace contains only trivial SCCs, and so there is no benefit in computing them. The graph of Figure 9.4 shows the time overhead of calculating the SCCs for this net, and Figure 9.5 shows the space overhead. (Note that the two graphs of disk space usage coincide, i.e. the disk space used when SCCs are computed is almost identical to the disk space used when SCCs are not computed.)

Although these overheads are relatively small, they may still be significant. Hence, in our implementation we provide the option to not calculate the SCCs (the definitions of the resulting RNSS can be obtained by treating the M^c notation to mean the state number of M rather than the SCC number of M). If the developer believes that the superplace reachability graphs will have few if any non-trivial SCCs (as may often be the case), they can use this option to eliminate the overhead. On the other hand, if the SCCs are non-trivial then the benefit of calculating the SCCs could far outweigh the overhead. It is an interesting avenue for future work to try to look for a heuristic that can be used to determine whether it is worthwhile computing the SCCs.

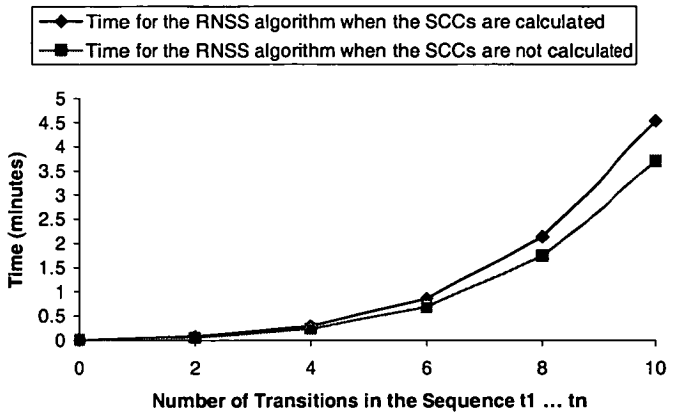


Figure 9.4: Time for computing SCCs for the net of Figure 9.1 (without t_{n+1})

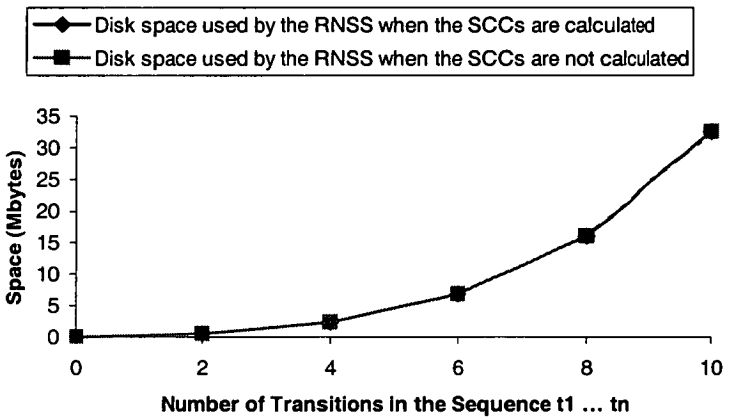


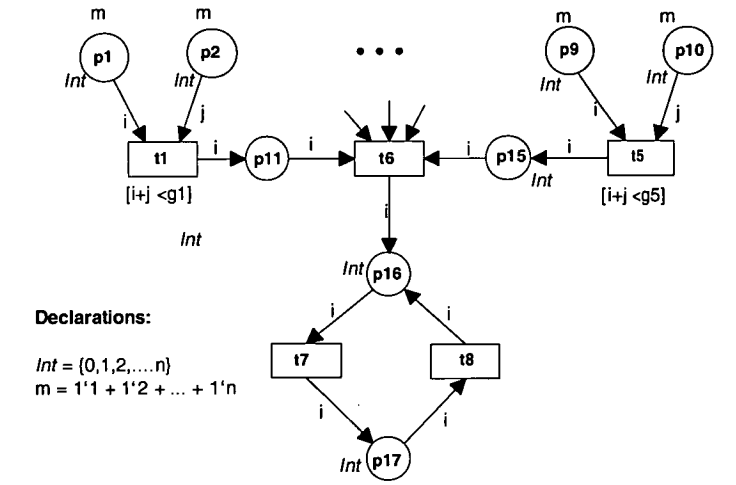
Figure 9.5: Space overhead for computing SCCs for the net of Figure 9.1 (without t_{n+1})

9.2 Net Properties Affecting Performance

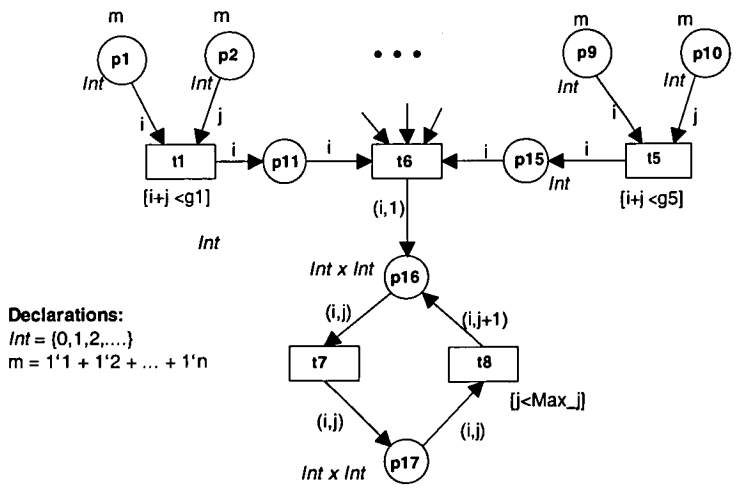
In the following sections we examine how various properties of the abstract and refined nets affect the performance of the type-subnet algorithm and the RNSS algorithm. We identify situations under which the incremental algorithms can be expected to yield performance improvement (compared to the standard algorithm), and the situations under which performance improvements are maximised. Since the algorithm that caters for all three forms of refinement is a combination of the type-subnet and RNSS algorithms then the observations made here also hold for that algorithm.

The implementation of the type-subnet algorithm uses the reachability graph of the abstract net to determine the enabled abstract firing modes, and hence the enabled refined firing modes (see Section 8.5.4). We indicate the time required to construct the abstract graph, together with the time required to construct the refined graph using the type-subnet algorithm, and refer to the sum of these as the *total time* for the type-subnet algorithm. In practice however, we would normally expect the reachability graph for the abstract net to be constructed and analysed before the refined net is considered. This would make the type-subnet algorithm even more attractive. As it is, in some situations even the total time for the type-subnet algorithm is less than the time required to construct the full reachability graph using the standard algorithm.

We demonstrate the first few properties of the type-subnet algorithm using the abstract net of Figure 9.6 (a), refined as shown in Figure 9.6 (b). In these nets the ellipses indicate the net contains another three subnets with the same structure as the subnet involving p_1, p_2, $p_1 1$, and t_1. This net is used purely because it allows us to easily demonstrate the various properties of the type-subnet algorithm. In both the abstract and refined net, the initial marking of each of the places p_1 to p_{10} is the multiset sum $1`1 + 1`2 + \ldots + 1`n$, where n is an integer. The abstract and refined nets have an initialisation section, where the tokens in the places p_1 to p_{10} are consumed. For example, the transition t_1 consumes a token from the place p_1 and a token from the place p_2, but the transition t_1 is only enabled when the sum of the tokens from p_1 and p_2 is less than g_1, where g_1 is an integer. As will be seen in the following sections, the net has been designed so that the various properties can be demonstrated by changing the value of the variables n and $g_1, \ldots, g_5$. By changing the value of n and $g_1, \ldots, g_5$ we are able to change the number of enabled and



(a)



(b)

Figure 9.6: An abstract net (a) and a refinement of it (b)

disabled firing elements that occur in the net. After the initialisation section, the abstract and refined nets have a processing section. Here tokens cycle between the places p_{16} and p_{17}. In the refined net, each token contains an integer value that is incremented on each iteration until the maximum value, Max_j, is reached.

In the following sections the parameters $n, g_1, \ldots, g_5$, and Max_j will be varied, and other changes will be made to demonstrate the various properties of the type-subnet algorithm. In particular we consider how the following properties affect the performance: the time taken to construct the abstract graph compared to that taken to construct the refined graph (Section 9.2.1); the number of disabled firing elements of the abstract and refined graphs (Section 9.2.2); the complexity of the arc functions of the abstract and refined graphs (Section 9.2.3); the amount of memory available (Section 9.2.4); the number of changed transitions of the refined net (Section 9.2.5); the amount of data stored in refined and non-refined tokens (Section 9.2.6); and the number of places and transitions added by subnet refinement (Section 9.2.7). We then consider how the amount of interleaving between internal activity of supernodes and external activity affects the performance of the RNSS algorithm (Section 9.2.8).

9.2.1 Time Taken to Construct the Abstract Graph

We first observe that the longer the standard algorithm takes to construct the refined graph relative to the abstract graph, then the greater the likelihood that the type-subnet algorithm will demonstrate improvements. This is shown in the graph of Figure 9.7, which plots the time to construct the refined graph using the standard algorithm and the total time taken for the type-subnet algorithm for the refined net of Figure 9.6(b), against the number of states of the refined graph. As the number of states of the refined graph increases, so does the time taken to calculate the refined graph. That is, the difference between the time for the abstract graph and that for the refined graph using the standard algorithm increases. In this example, to increase the number of states of the refined graph we increase Max_j. Hence the number of states of the refined graph is increased by type refinement. Alternatively, we could have increased the number of states using subnet or node refinement, or a combination of all three refinements. For these results, the value of n is set to 10, the variables

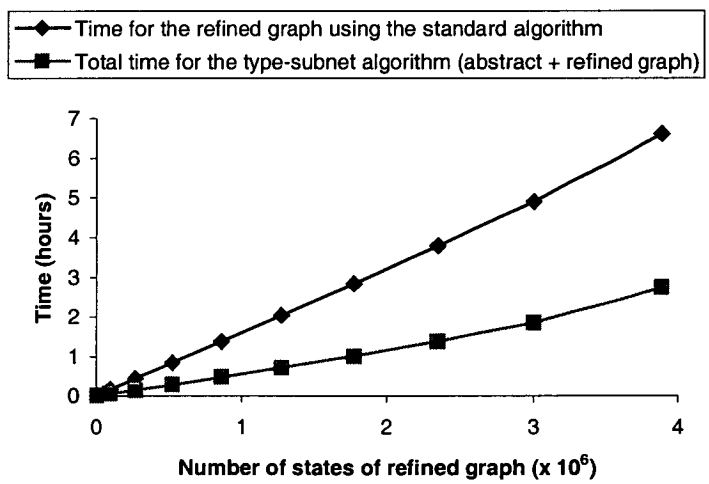


Figure 9.7: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of states of the refined graph increases

$g_1 = \cdots = g_5 = 4$, and the abstract graph takes 30 seconds to construct. Virtual memory is being used by both algorithms when the refined graph is greater than 3×10^6 states, and all virtual memory is used when we try to generate graphs larger than the largest result shown. We consider memory usage and its effects on the performance of the type-subnet algorithm further in Section 9.2.4.

9.2.2 Number of Disabled Firing Elements

One significant advantage gained by the type-subnet algorithm is that refined firing elements do not have to be considered if the corresponding abstract firing element is disabled. Therefore we can expect good performance improvement for the type-subnet algorithm compared to that of the standard algorithm if there is a large number of refined firing elements for which the corresponding abstract firing element is disabled. By changing the value of n and $g_1, \ldots, g_5$ in the nets of Figure 9.6 (a) and (b), we can vary the number of disabled firing elements in the initialisation and so demonstrate this effect. The graph of Figure 9.8 plots the time taken for the refined graph using the standard algorithm, the time for the abstract graph (using the standard algorithm), and the total time for the type-subnet algorithm, as the value of n is increased in the abstract and refined net of Figure 9.6 (a) and (b) respectively (with the value of Max_j set to 100, and the values of $g_1 = \cdots = g_5 = 4$). Since only the tokens input to the transition t_i, whose sum is less than g_i, are enabled, then increasing the value of n in the initial marking has the effect of increasing the number of disabled firing elements.

The number of refined firing elements is due not only to the number of tokens (as was varied in the above example), but also to the number of transitions. A similar effect can

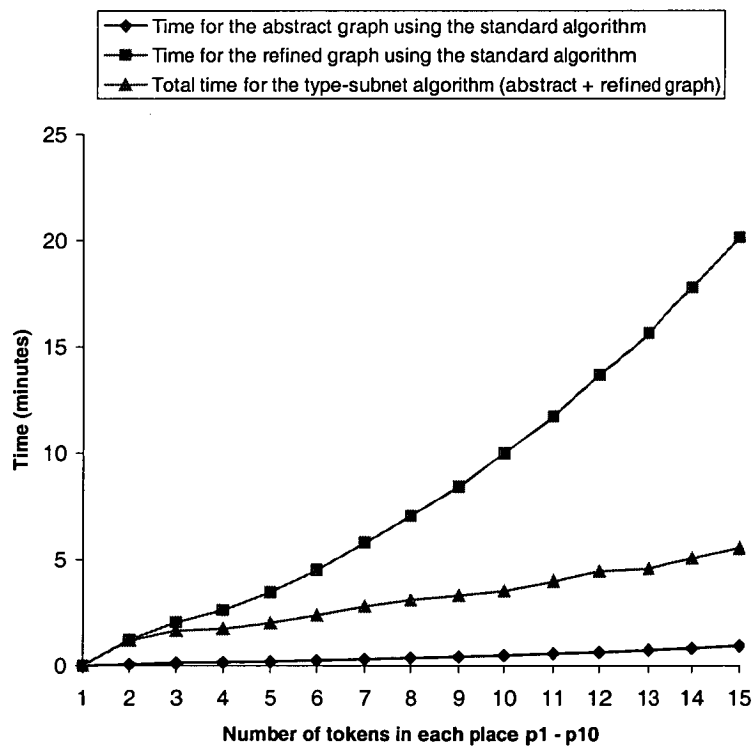


Figure 9.8: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of disabled firing elements is increased

therefore be observed by increasing the number of disabled firing elements in the abstract and refined graph by adding transitions to both the abstract and refined nets.

9.2.3 Calculating Successors

In the previous section we demonstrated that the type-subnet algorithm can lead to performance improvement since the refined firing element does not have to be considered if the corresponding abstract firing element is disabled. Another advantage of the type-subnet algorithm is that if the corresponding abstract firing element is enabled, and the transition has not been changed in the refinement, then the successor marking due to that transition can be calculated from the abstract successor marking. This is done using the UPDATE function described in Chapter 7.

Depending on the amount of change between the abstract and refined net, the number of input and output arcs of the transition, and the complexity of the expressions on the arcs, it may actually be more efficient to calculate the input and output effect of the transition. However, if the arc expressions are complex, then it is likely that updating the abstract successor will be more efficient.

We can demonstrate this by changing some of the arc expressions in the nets of Figure 9.6. The current output arc expressions for the transitions t_1 to t_5 of the nets of Figure 9.6 are simple to evaluate. We change these expressions to an expression that evaluates as described in function of Figure 9.9. This new arc expression will evaluate to the same as the original expression, but can take considerably longer to evaluate depending upon the value of *max*. Clearly in practice it would not be sensible to replace the original arc expression with the modified one. However, by changing the value of *max*, we change the time required to compute the input and output effect of the transition, and therefore demonstrate a potential benefit of using the UPDATE function.

```
F(i)
begin
  Result := ∅
  for all k ∈ {1..max}
    if k = i
      Result := Result + k
    end if
  end for
  return Result
end
```

Figure 9.9: Modified output arc function

The graph of Figure 9.10 shows the time for the abstract graph, the total time for the type-subnet algorithm, and the time for the refined graph using the standard algorithm as the value of *max* in the function of Figure 9.9 is increased[1]. In this example, the value of

[1]The function of Figure 9.9 was implemented in Maria using quantification, since iteration in the form of *for*, *while*, and *repeat* loops is not supported.

n was 10, $Max\_j = 5$ the value of $g_1 = g_2 = g_3 = 4$, and $g_4 = g_5 = 5$.

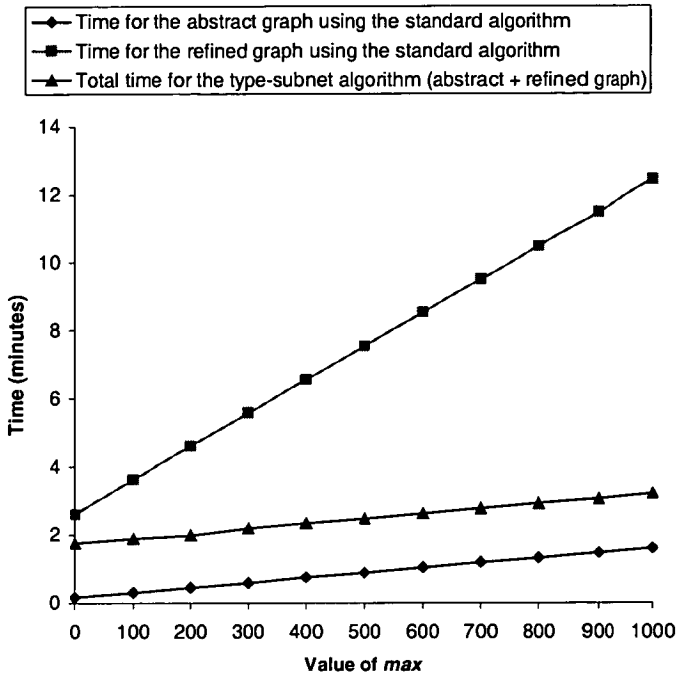


Figure 9.10: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the complexity of the arc functions is increased

9.2.4 Memory Usage

In the implemented version of the type-subnet algorithm, the system memory used is greater than that for the standard algorithm since a representation (hash table) of the abstract graph is stored in system memory (RAM) by the type-subnet algorithm. If the size of the hash table of the abstract graph together with the hash table of the refined graph is greater than the available system memory, then the performance of the type-subnet algorithm will suffer since virtual memory and the associated page swapping is significantly slower than system memory. However, as is shown in the graph of Figure 9.7, even when virtual memory is used, the type-subnet algorithm may still be significantly faster than the standard algorithm.

The effect of memory usage is shown in the graph of Figure 9.11, which plots the total time for the type-subnet algorithm, the time to construct the abstract graph, and the time taken by the standard algorithm to construct the refined graph as the amount of available system memory (RAM) is changed for the nets of Figure 9.6. To change the amount of RAM available, we simply allocate memory without deallocating it at the start of the program. (Recall that the machine being used for testing has 256 Mbytes of RAM.) In this example, the value of n was 4, $Max\_j = 100$ the value of $g_1 = g_2 = g_3 = g_4 = g_5 = 4$.

Due to the overhead of the operating system kernel swapping pages to and from disk, and the associated unpredictable time delays, if available RAM is low the time taken for each algorithm to run can vary. Since in this example we are only concerned with observing the general trend of the time taken by the algorithms, this is not a major concern. To

obtain results that better reflect the trend, each value plotted in the graph of Figure 9.11 is the average of five separate runs.

The graph shows that the improvement of the type-subnet algorithm sharply increases when the amount of available RAM is approximately 40 Mbytes. By monitoring the memory used, we observe that this is when the kernel stops using virtual memory. That is, the size of the hash table of the abstract graph, the hash table of the refined graph, and the memory used by the kernel, is approximately 40 Mbytes. When there is less than 40 Mbytes of memory available the time of the type-subnet algorithm increases due to the extra time taken to read from and write to virtual memory. The time of the standard algorithm does not increase even when it requires virtual memory, because it does not perform as many operations requiring data to be read from memory.

The parameters of the nets in this example had to be chosen very carefully so that the size of the refined graph was similar to the size of the abstract graph. If the refined graph is much larger than the abstract graph, then the amount of extra memory used to store the abstract graph becomes insignificant (see the graph of Figure 9.7).

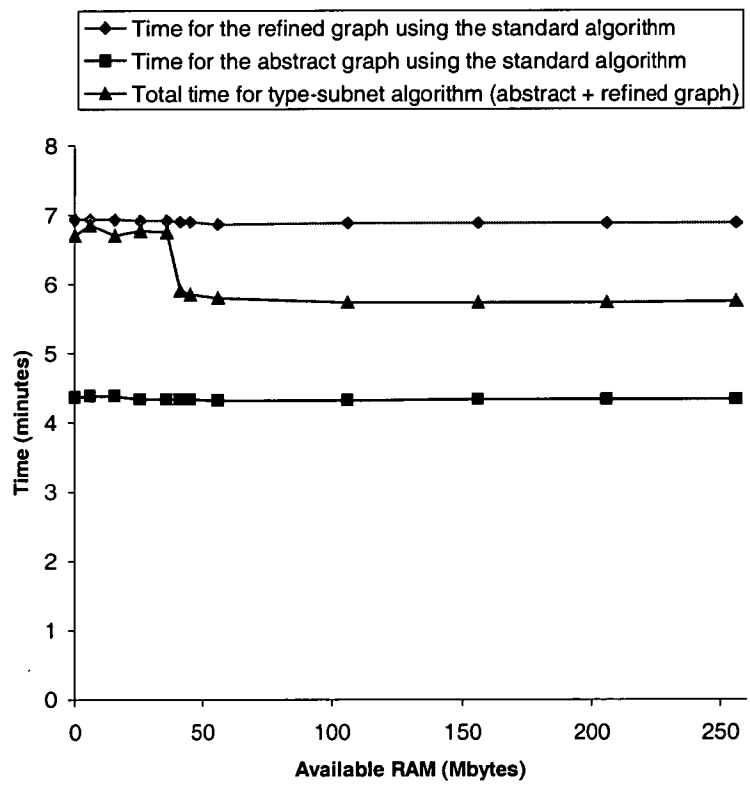


Figure 9.11: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of available RAM is increased

9.2.5 Number of Changed Transitions

Recall that a transition is said to be *changed* by type or subnet refinement if the type of its neighbouring places has been refined, or if it has arcs added or existing arcs modified from its corresponding abstract version (see the function CHANGED-TYPESUBNET in Section 7.5). As was discussed in Section 8.5.1, the version of Maria used to implement the

type-subnet algorithm is such that (in the context of markings) the refined token elements that map to a given abstract token element cannot be efficiently found. This has meant that in the implemented type-subnet algorithm (Algorithm 8.2), if a transition has been changed by type or subnet refinement then the enabled firing modes and corresponding successors due to the transition are found using the same method as the standard algorithm. On the other hand, if a transition has not been changed then the enabled firing modes can be found directly from the abstract graph and the refined successor marking can be found by updating the abstract successor marking.

Therefore we expect that, in our implementation, the fewer changed transitions the better the performance of the type-subnet algorithm. The graph of Figure 9.12 shows the time of the standard algorithm and the total time for the type-subnet algorithm as we change transitions by refining the type of their neighbouring places (places p_1 to p_{15}) in the net Figure 9.6 (b). In this example the value of Max_j is set to 100, the value of n is 4 and $g_1 = \cdots = g_5 = 4$. The value, x, on the horizontal axis of the graph, indicates that the places p_1 to p_x together with p_{16} and p_{17} have been refined by type refinement. For example, the value 6 indicates that places p_1 to p_6 and places p_{16} and p_{17} have been refined by type refinement. As another example, the value 0 indicates that only the places p_{16} and p_{17} have been refined. (To avoid the effect of adding data by type refinement (see Section 9.2.6) the type refinement used here does not add any extra data to the abstract type. That is the data stored by the refined type and the abstract type is the same.) Similar results to the graph of Figure 9.12 can be obtained by changing transitions by extending the type of connected places, or by adding arcs to the transition.

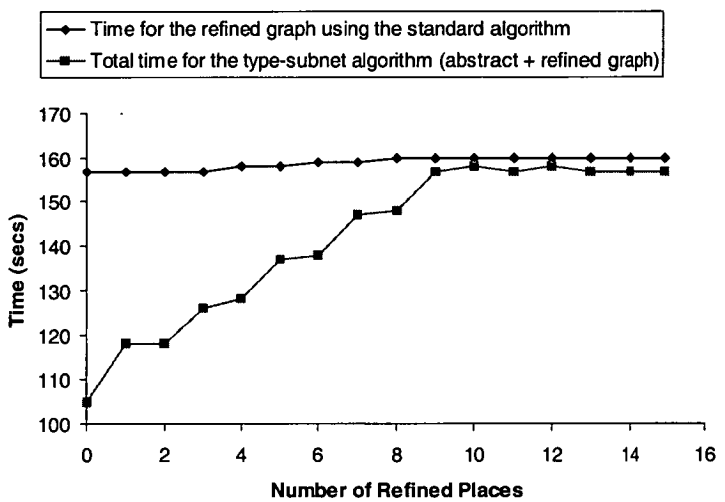


Figure 9.12: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of refined places in the net of Figure 9.6 (b) is increased

It can be seen in the graph of Figure 9.12 that the difference between the total time for the type-subnet algorithm and the time for the standard algorithm decreases as the number of places that are refined increases. As can be expected, if the type of the place p_1 is refined then the transition t_1 is *changed*, and so refining the type of place p_2 as well has little effect on the performance of the type-subnet algorithm. This also applies to the place p_4 with transition t_2, place p_6 with transition t_3, etc.

The amount of decrease in performance due to changing a transition will depend on how much advantage the type-subnet algorithm gains from the transition before it is changed. That is, it will depend on the number of modes of the transition that would normally be checked for enabling, but do not need to be checked when using the type-subnet algorithm. For example, unlike the transitions t_1 to t_5, the majority of modes of the transitions t_7 and t_8 are enabled (since the transition t_7 has no guard, and the majority of modes for t_8 will satisfy its guard). Therefore the type-subnet algorithm is not significantly faster than the standard algorithm for computing the enabled modes and successors of t_7 and t_8. Hence the type-subnet algorithm does not decrease in performance when the transitions t_7 and t_8 are changed by refining their neighbouring places.

It is interesting that in some other examples we observed a rather unexpected result of the refined graph being constructed in less time for both the standard algorithm and the type-subnet algorithm as more refinements were made to the refined net. That is, we observed behaviour opposite to the behaviour demonstrated above. After careful investigation including profiling the code, we discovered this was because the refinement added data that meant fewer collisions occurred in the hash table used to represent the graph of the refined net in system memory. The more refinements that were made to the net, the more collisions were eliminated. This makes it apparent that the data structures and state storage mechanisms used (such as the hash table) can have subtle interactions with the performance of reachability analysis. For example, the use of hash functions to retrieve prior states can be affected by various things such as a possible conflict between the hash function and the structure of the states being hashed.

We have shown that we can expect better improvement from the type-subnet algorithm if the number of changed transitions is kept to a minimum. Thus when faced with the decision of whether to refine the net by extending the colour of a transition (and possibly its neighbouring places), or by adding new places and transitions (i.e. a partial unfolding of the first option), then the second option is likely to be more efficient[2]. This is because it does not involve changing abstract transitions. For example, suppose in addition to the refinement shown in Figure 9.6 (b), we wish to add a new token of value $n + g_i$ to each of the places p_1 to p_{10}, and transitions to handle these new tokens. This can be achieved by subnet refinement in two ways: first as shown in Figure 9.13, or second as shown in Figure 9.14. Figure 9.14 extends the token type of existing places while Fig 9.14 adds new places to hold these new token values. The unfolding of the two nets is identical. With values $n = 4$, $g_1 = g_2 = g_3 = 3$, $g_4 = g_5 = 4$ and $Max\_j = 100$, for the net of Figure 9.13, the standard algorithm takes 230 seconds, and the total time for the type-subnet algorithm is 280 seconds (where the abstract graph takes less than 1 second). On the other hand, for the net of Figure 9.14 the standard algorithm takes 198 seconds and the total time for the type-subnet algorithm is 143 seconds (where the abstract graph takes less than 1 second). That is, the type-subnet algorithm does not lead to performance improvement when the refinements are made by extending the type, but does lead to improvement when

[2]Unless the second option requires many more transitions than the first.

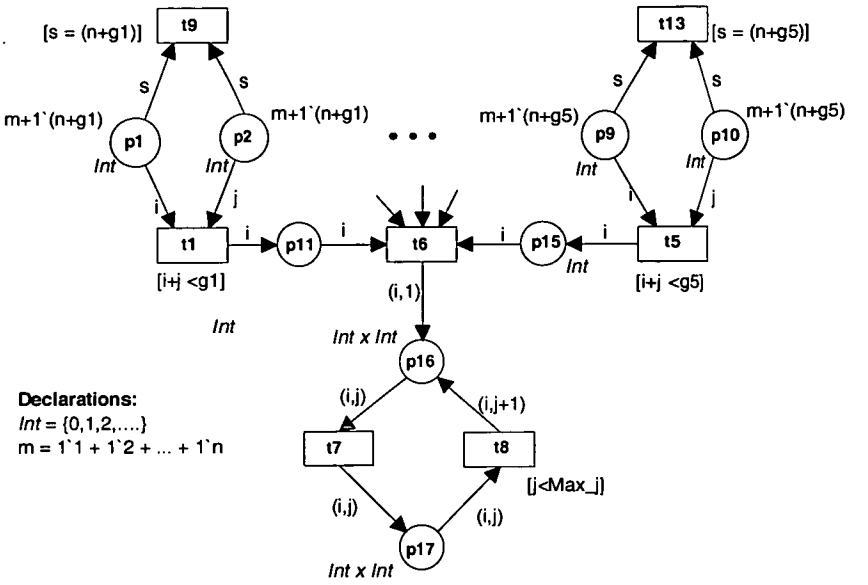


Figure 9.13: One refinement of Figure 9.6 (a)

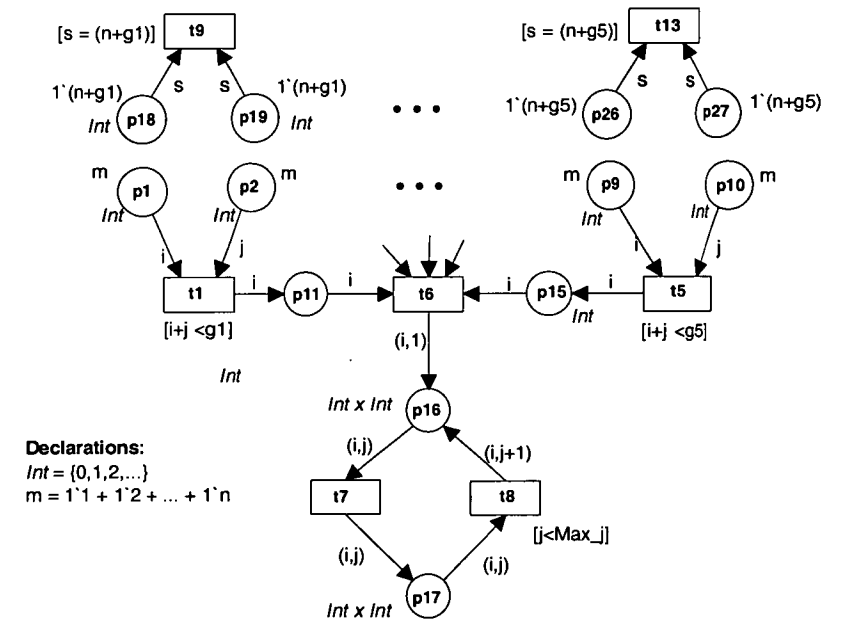


Figure 9.14: Another refinement of Figure 9.6 (a)

the same effect is achieved by adding places and transitions. When refinements are made by extending the type, the extra time taken by the type-subnet algorithm compared to the standard algorithm can be explained by the overhead of the type-subnet algorithm: in the type-subnet algorithm the abstract edges are found (from the abstract reachability graph) before transition instance analysis. The standard algorithm does not require this computation before transition instance analysis.

9.2.6 The Amount of Data Stored in the Tokens

We now consider how the amount of data stored in the tokens affects the performance of the type-subnet algorithm compared to that of the standard algorithm. We note that the effects discussed here are specific to the state storage used in Maria (Section 8.1.3), and may differ if another state storage mechanism is used.

As was explained in Section 8.1.3, the standard reachability graph algorithm encodes every marking to a bit-string and it is this bit-string which is stored. The bit-string is then decoded when the marking is retrieved. Hence every time the standard algorithm retrieves a marking (to find the successors from that marking) it must first decode the bit-string representation of the marking. On the other hand, the UPDATE function of the type-subnet algorithm can update the bit-string representation directly without first decoding it, saving the time required to decode the source marking and then encode the successor marking. Therefore we would expect that an increase in the amount of data stored in the tokens will result in improved performance of the type-subnet algorithm compared to the performance of the standard algorithm. This is demonstrated in the graph of Figure 9.15, where strings of 10 characters have been added to the tokens of the nets in Figure 9.6 (a) and (b). Once again $g_1 = \cdots = g_5 = 4$, and $Max\_j = 100$. The x-axis value indicates the number of strings that have been added.

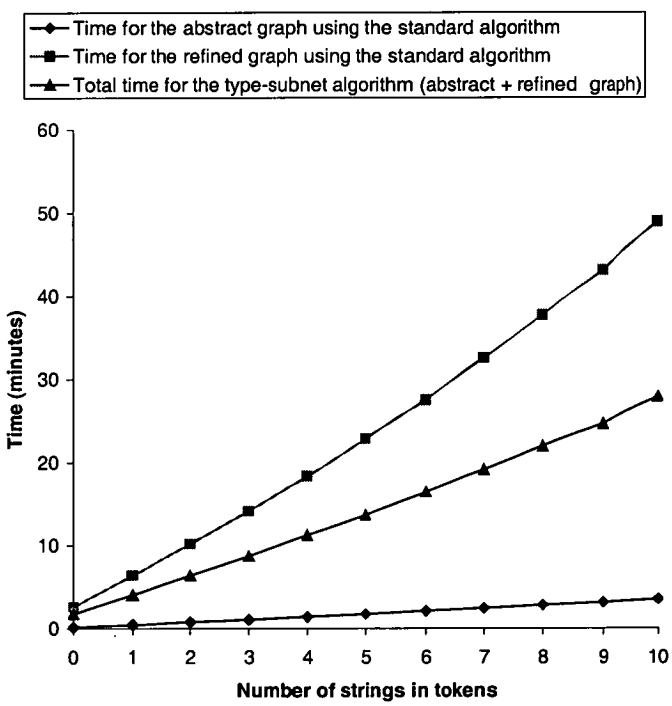


Figure 9.15: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of data in non-refined tokens is increased

It is also useful to consider how the amount of data added by type refinement affects the performance of the type-subnet algorithm compared to that of the standard algorithm. We have already discussed the effect of type refinement where the size of the refined graph is increased (Section 9.2.1), so here we consider adding data by type refinement where the size of the refined graph is not affected. Since the type-subnet algorithm must decode and encode markings of places that have been refined, adding data by type refinement will not yield performance improvement similar to the above. However, the standard algorithm must also decode and encode the extra information stored in the tokens, and so the time improvement of the type-subnet algorithm should remain the same. This is demonstrated in the graph of Figure 9.16, where the type of places p_{16} and p_{17} are refined to include a number of strings, each string being 10 characters long. (The value of the x-axis indicates the number of strings in the refined type.)

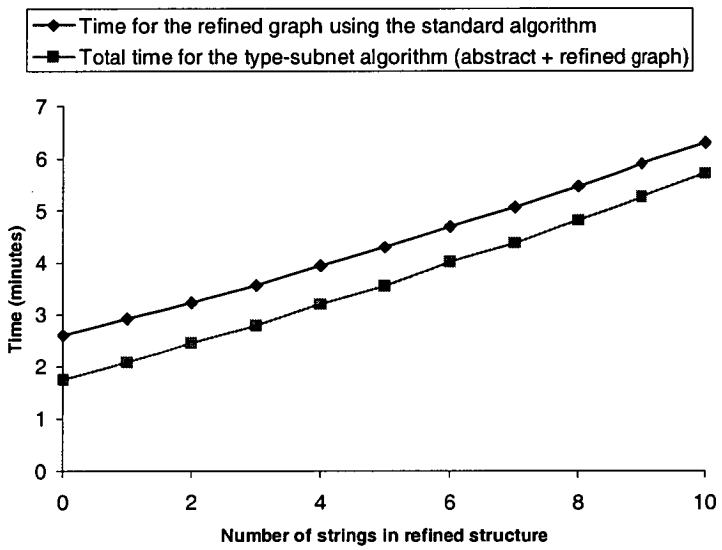


Figure 9.16: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of data in refined tokens is increased

9.2.7 Number of Places and Transitions Added by Subnet Refinement

As we have described in Section 9.2.1, as the difference between time taken to construct the abstract graph and the time taken to construct the refined graph using the standard algorithm increases, then so does the potential for significant performance improvement. We now consider the effect of adding places and transitions using subnet refinement, where the additions do not change the size of the graph. Since the transitions are newly added, then the type-subnet algorithm cannot use the abstract net to help determine the enabled refined modes involving these transitions. Hence the standard and the refined algorithm both find the enabled firing modes of the newly added transitions using the same method (in Maria this is the transition instance analysis algorithm), and the amount of improvement of the type-subnet algorithm remains the same. This is demonstrated in the graph of Figure 9.18, which plots the performance of the type-subnet algorithm compared to that of the standard algorithm for the net of Figure 9.6(a), refined as shown in Figure 9.17 (where $g_1 = \cdots = g_5 = 4$, and $Max\_j = 100$). This net has a number of added transitions, each of which consumes from place p_{18} (which is empty) and generates tokens in p_{19}.

(Figure 9.17 has 5 transitions added). The value on the x-axis indicates the number of transitions added. A similar result can be obtained by adding places or a combination of places and transitions.

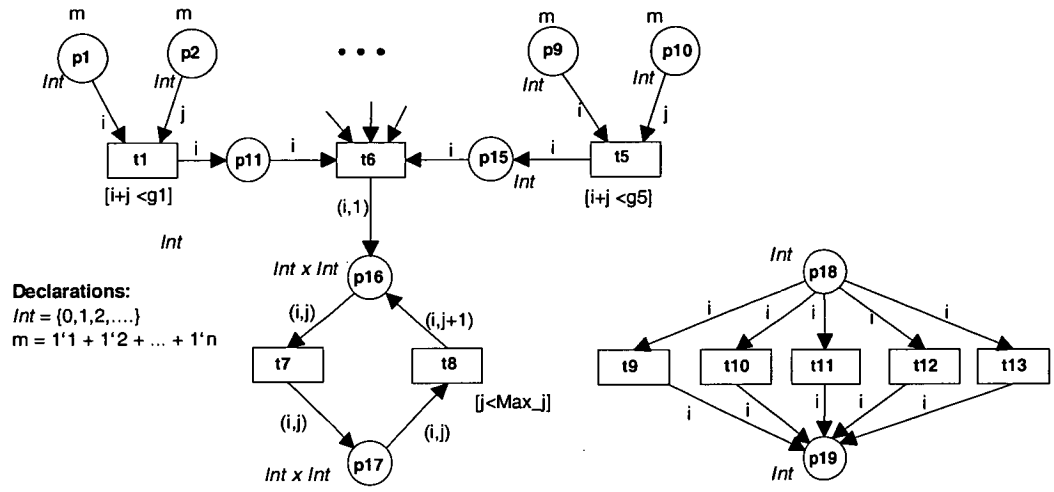


Figure 9.17: The net of Figure 9.6 (a) refined by subnet refinement

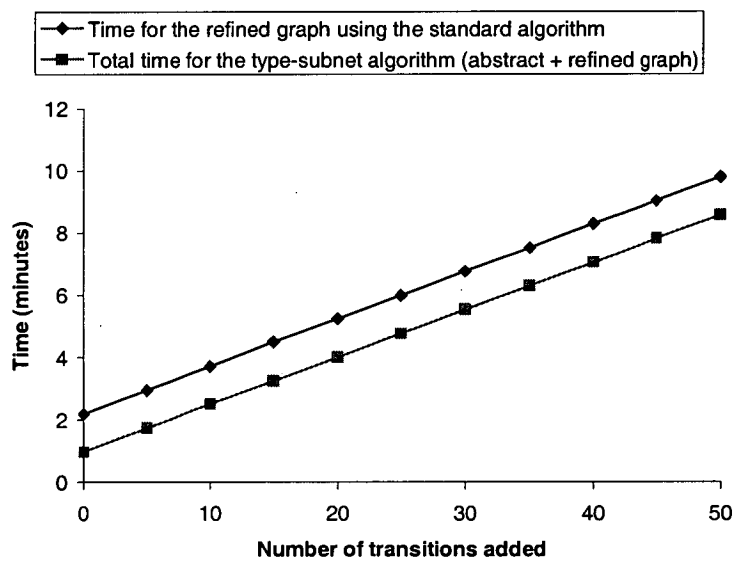


Figure 9.18: The performance of the type-subnet algorithm on the nets of Figure 9.6 (a) and Figure 9.17 as transitions are added to the net of Figure 9.17

9.2.8 RNSS Algorithm

A significant advantage of the RNSS algorithm is that for a net with supernodes it can save time and space because it will not consider all the possible interleavings of the internal activity of the supernodes and external activity. As the extent of interleaving between internal and external transitions of the net increases, so does the performance improvement of the RNSS algorithm, compared to that of the standard algorithm. Here we consider the

performance of the RNSS algorithm. We demonstrate this effect using both superplaces and supertransitions.

The superplace example is the net shown in Figure 9.19. The supertransition example is shown in Figure 9.20. These nets consist of a superplace (supertransition) and the places ep_1, ep_2, and ep_3, and transitions et_1 and et_2, where et_1 consumes from ep_1 and generates tokens in ep_2, and et_2 consumes from ep_2 and generates tokens in ep_3. The superplace (supertransition) consists of a canonical basis together with a sequence of n transitions and corresponding places (from p_1 to p_{n+1}). The amount of internal activity in both the superplace example (Figure 9.19) and the supertransition example (Figure 9.20) is dependent on the number of transitions in the internal sequence. That is, the amount of internal activity is dependent on the value of n. (Note that both the superplace and supertransition in the example consist of a canonical basis together with a sequence of internal transitions. Hence the number of places and transitions in the superplace is different to the number in the supertransition.)

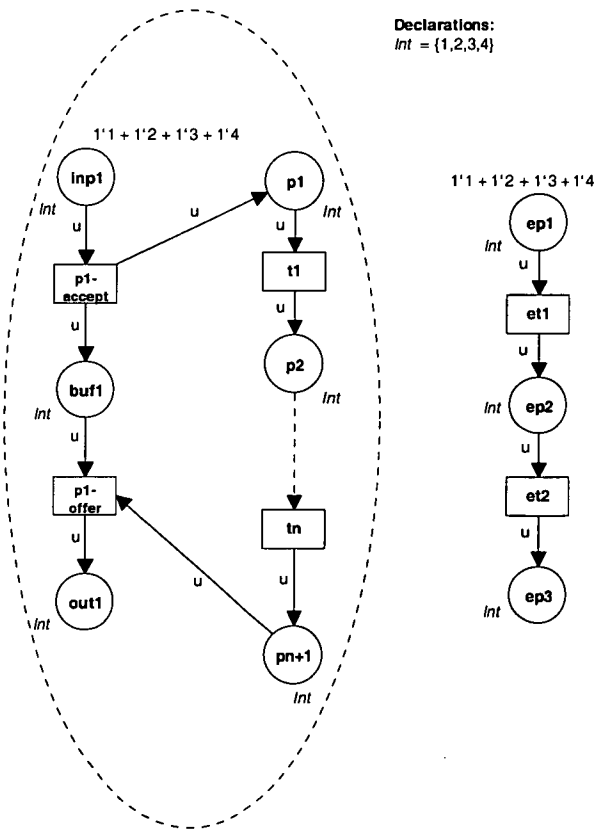


Figure 9.19: A net with a superplace

The graphs of Figure 9.21 and Figure C.16 show the time performance of the RNSS algorithm compared to that of the standard algorithm as the level of internal activity is

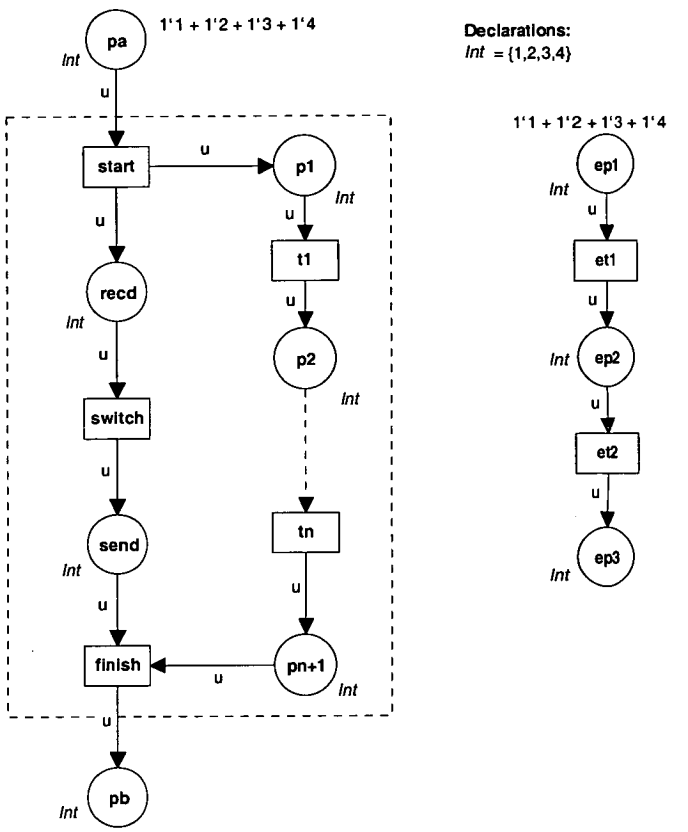


Figure 9.20: A net with a supertransition

increased in the superplace and supertransition examples respectively. In each case the x-axis indicates the value of n.

Here we observe that when there were 13 or more transitions in sequence in the superplace, the standard algorithm could not compute the full reachability graph since it ran out of virtual memory. Similarly, when there were 7 or more transitions in sequence in the supertransition, the standard algorithm could not complete due to insufficient virtual memory. On the other hand, the RNSS algorithm constructed the RNSS in each example in only a few minutes. (The discrepancy between the number of transitions possible in the sequence of the superplace example and the number of transitions possible in the sequence of the supertransition example is due to the fact that superplaces and supertransitions have difference canonical bases).

Since the total number of vertices and edges in the RNSS is significantly fewer than the number of vertices and edges in the full graph, then the amount of memory used by the RNSS algorithm is less than the amount of memory used by the standard algorithm. The graphs of Figure 9.23 and Figure 9.24 plot the total disk space used for the RNSS compared to that used for the standard algorithm for the superplace and supertransition examples respectively.

The amount of internal activity in the superplace (supertransition) is due not only to the number of internal transitions, but also the number of tokens in the superplace (available to the supertransition). We therefore obtain similar results to the graphs of Figures 9.21 – 9.24 by increasing the number of tokens in the superplace. Since the RNSS algorithm eliminates interleaving between internal and external activity, it also shows similar perfor-

mance improvement as the amount of activity external to the supernode is increased, and as the number of supernodes in the net is increased.

The graph of Figure 9.25 shows how the time taken by the RNSS algorithm to construct the RNSS compares to the standard algorithm for a net consisting of a number of copies of the superplace shown in Figure 9.19 (the external places ep_1 - ep_3 and transitions et_1 and et_2 are not included). The x-axis indicates the number of copies of the superplace. For these results the initial marking of the place *inp1* of each superplace had to be set to $1`1 + 1`2 + 1`3$. (Note the initial marking is different to that shown in Figure 9.19 since the token $1`4$ is not present.) The value of n was 2. Here the standard algorithm could not compute the full reachability graph for a net consisting of four or more copies of the superplaces, due to insufficient virtual memory. The disk space used by the RNSS compared to that of the standard algorithm as the number of copies of the superplace is increased is shown in Figure 9.26. With an extra token $1`4$ added to the place *inp1* of the superplace, so that the initial marking was $1`1 + 1`2 + 1`3 + 1`4$ (as it appears in Figure 9.19), and $n = 2$, the standard algorithm could not even complete for a net consisting of two copies of the superplace, again due to insufficient virtual memory.

The graphs of Figure 9.27 and 9.28 show how the time and space taken by the RNSS algorithm to construct the RNSS compares to that of the standard algorithm for a net consisting of a number of copies of the supertransition shown in Figure 9.20. Each copy of the supertransition consumes from p_a and generates tokens in p_b (the external places ep_1 - ep_3 and transitions et_1 and et_2 are not included). The x-axis indicates the number of copies of the supertransition. For these results the initial marking of the place p_a was $1`1 + 1`2 + 1`3 + 1`4$. The value of n was 2. Here the standard algorithm could not compute the full reachability graph for a net consisting of eight or more copies of the supertransition, due to insufficient virtual memory.

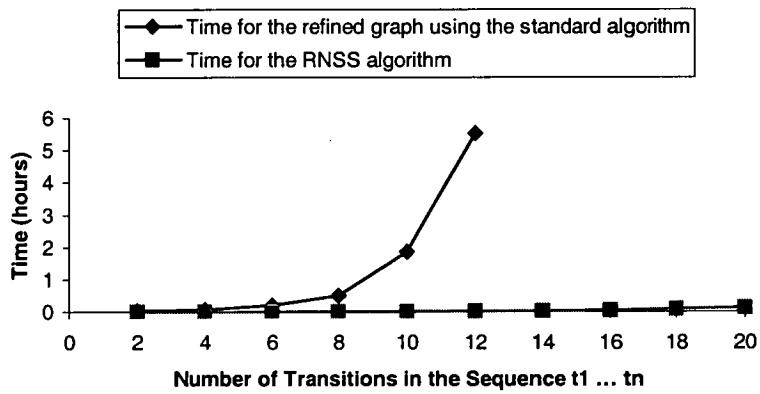


Figure 9.21: The time performance of the RNSS for the superplace example (Figure 9.19)

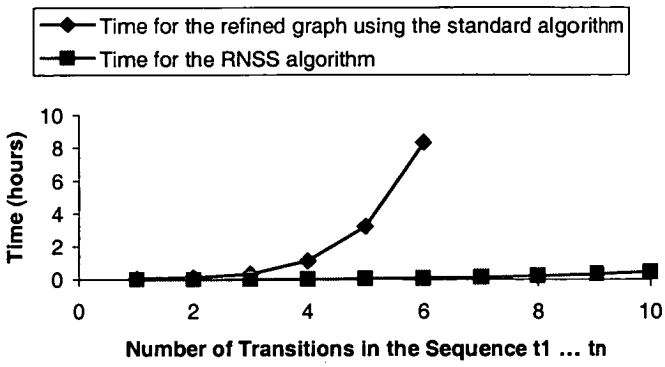


Figure 9.22: The time performance of the RNSS for the supertransition example (Figure 9.20)

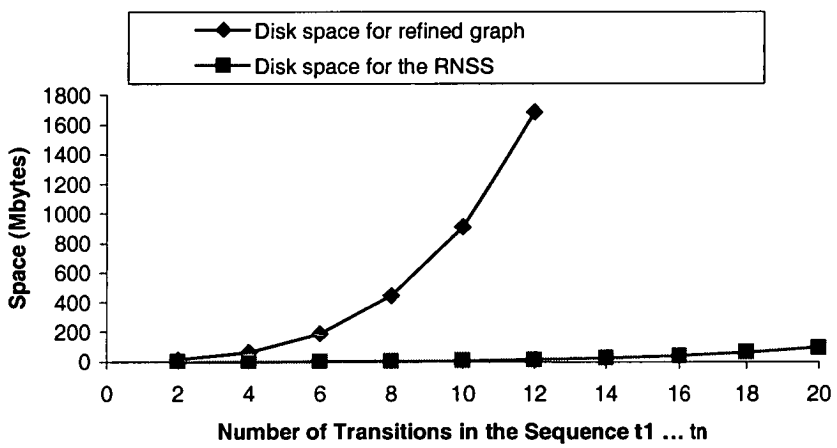


Figure 9.23: The disk space performance of the RNSS for the superplace example (Figure 9.19)

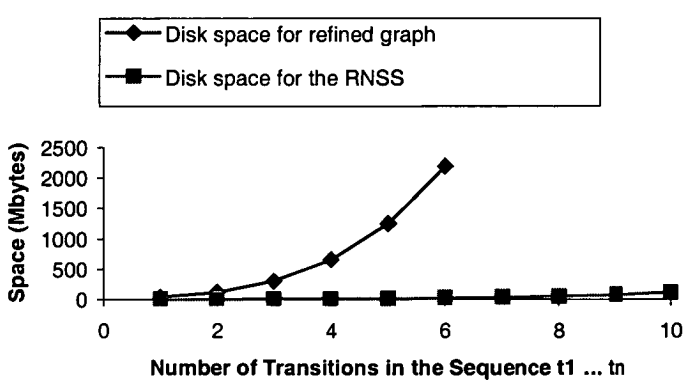


Figure 9.24: The disk space performance of the RNSS for the supertransition example (Figure 9.20)

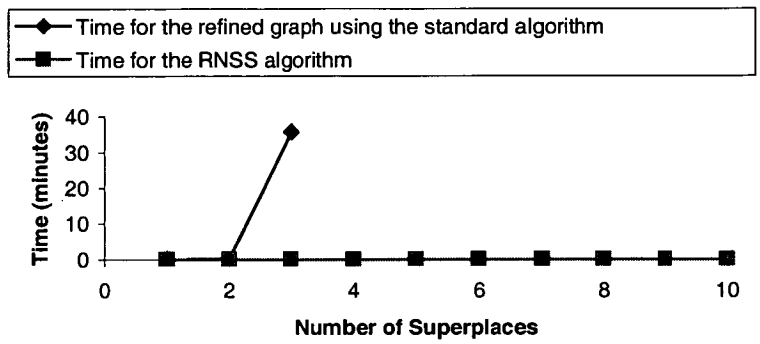


Figure 9.25: The time performance of the RNSS for a net with several copies of the superplace of Figure 9.19

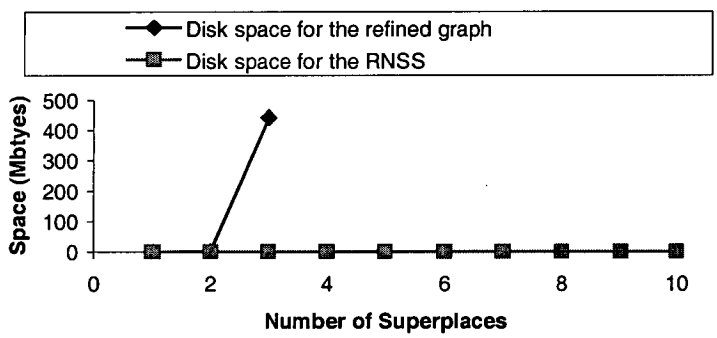


Figure 9.26: The disk space used by the RNSS for a net with several copies of the superplace of Figure 9.19

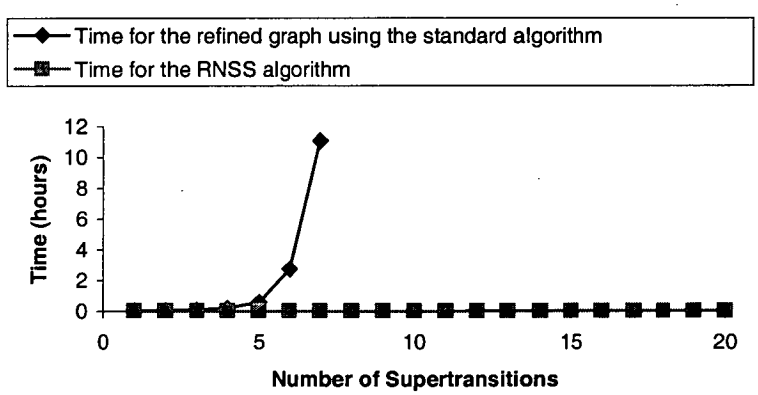


Figure 9.27: The time performance of the RNSS for a net with several copies of the supertransition of Figure 9.20

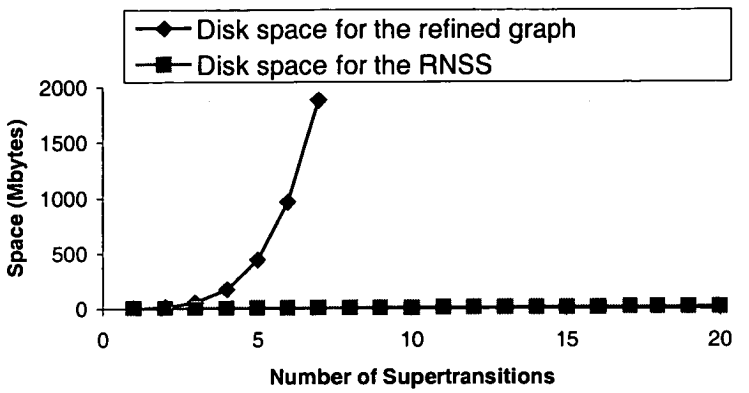


Figure 9.28: The disk space used by the RNSS for a net with several copies of the supertransition of Figure 9.20

9.2.9 Summary

As demonstrated by the preceding sections, the algorithm that caters for type and subnet refinement will generally be faster than the standard algorithm if:

- there is a large number of refined firing elements for which the corresponding abstract firing elements are disabled.
- the functions associated with the arcs of the abstract graph are complex (in terms of time to compute them).
- the tokens of the abstract and refined net hold a large amount of data.

Both the abstract and refined graph must be represented in system memory. If the abstract graph is large, then virtual memory may be required, and the performance of the type-subnet algorithm will suffer. However, if the refined graph is much larger than the abstract graph, then the amount of extra memory used to store the abstract graph becomes insignificant. Also, the performance improvement of the type-subnet algorithm compared to the standard algorithm will decrease as more transitions of the abstract net are changed in the refined net. If the size of the refined reachability graph remains the same when data is added by type refinement or places and transitions are added by node refinement, then time improvement of the type-subnet algorithm compared to that of the standard algorithm will remain the same.

We have also observed that the performance of the RNSS algorithm compared to the standard algorithm improves as the interleaving between the internal transitions of supernodes and external transitions or transitions internal to other supernodes increases. Further to this, if there is a time saving by constructing the RNSS rather than the complete graph, then there is a corresponding space saving. We have observed that for artificial examples, the RNSS algorithm could produce the state space in a matter of seconds, as opposed to the standard algorithm which could not complete. Since the algorithm that caters for all three forms of refinement is a combination of the type-subnet algorithm and the RNSS algorithm then the situations that lead to performance improvement in each of these algorithms (as described above) will also lead to improvement in the combined algorithm.

To assess the performance of the incremental algorithms in practice, we now examine two separate case studies: the Z39.50 protocol (Section 9.3) and the Missile Simulator (Section 9.4).

9.3 Z39.50 Protocol

As discussed in Section 5.2.2, the Z39.50 Protocol for Information Interchange [13] has been modelled incrementally by Lakos and Lamp [127, 123, 122]. In the following sections, we first consider the implementation of the basic Z39.50 model (Section 9.3.1), and then the performance of the type-subnet algorithm for: the model refined with Segmentation (Section 9.3.2), the model refined with Access Control (Section 9.3.3), and the model refined with both Segmentation and Access Control (Section 9.3.4).

9.3.1 Implementing the Basic Z39.50 Model

As described in Section 5.2.2, Lakos and Lamp [122] use a particular style of subnet to specify origin-initiated request services of the Z39.50 protocol, including: *Initialize*, *Release*, *Search*, *Present*, *Delete*, and *Resource-report*. This subnet is referred to as a *request* subnet (see Figure 5.4). Target-initiated response services of the Z39.50 protocol are modelled using a similar subnet, referred to as the *response* subnet. The Z39.50 origin can be implemented by using an instance of the request subnet (Figure 5.4) for each of the different requests, and the Z39.50 target can be implemented by instantiating the response subnet for each of the different responses. In this way, the sending and receiving of each message is handled by a separate transition. The Z39.50 Protocol can be implemented by instantiating the origin and target in a suitable environment.

We have implemented the basic Z39.50 protocol in Maria. The implementation consists of a client and origin, shown in Figure 9.29 and a server and target, shown in Figure 9.30. In these figures dashed boxes have been placed around the client, origin, server, and target so each can be easily identified. The client is modelled simply as a repository of message requests and transitions that send those requests. Similarly the server is modelled as a repository of responses and transitions that send those responses. A transition (not shown) from the *originToNet* place to the *targetFromNet* place transfers requests from the origin to the target. Similarly a transition (not shown) from the *targetToNet* place to the *originFromNet* place transfers responses from the target to the origin.

The state places (*originState*, *serverState*, and *targetState*) indicate whether the origin, server, or target, are *open* or *closed*, that is, whether the origin, server, or target is in a state where it can send and receive requests and responses. The sent and received places (*originSent* and *targetReceived*) store an indication of messages that have been sent and received respectively. All other places store actual requests and responses, and have the colour *Message*. To avoid clutter, we have not included the arc inscriptions or colours in the figures.

Each token of type *Message* can be either an *application data unit* or a *protocol data unit*. The information stored by these data units is specified in the Z39.50 protocol definition. To model the basic operation of the protocol, the messages do not need to contain all the information specified by the protocol. The model therefore uses a minimal message format which can be refined at a later stage by type refinement.

The initial marking of the client contains all the requests, while the initial marking of the server contains all the responses. In the initial marking shown, the client message repository contains one of each type of request (*Initialize*, *Search*, *Present*, *Release*) and an *Abort* signal. The server contains responses for each of these requests and an *Abort*

signal of its own. We believe this marking is typical of a scenario that will be analysed using reachability analysis.

We have simply used a field in the message to match requests with responses, that is, each request token contains the identity of the response that is expected. Note that in the initial marking shown there are three *Present* responses. This models the case where not all records fit in a single response message. The first *Present* response will be sent due to the initial *Present* request, while the other *Present* responses will be due to the *Present* request generated by the *generateNextPresentRequest* transition of the client.

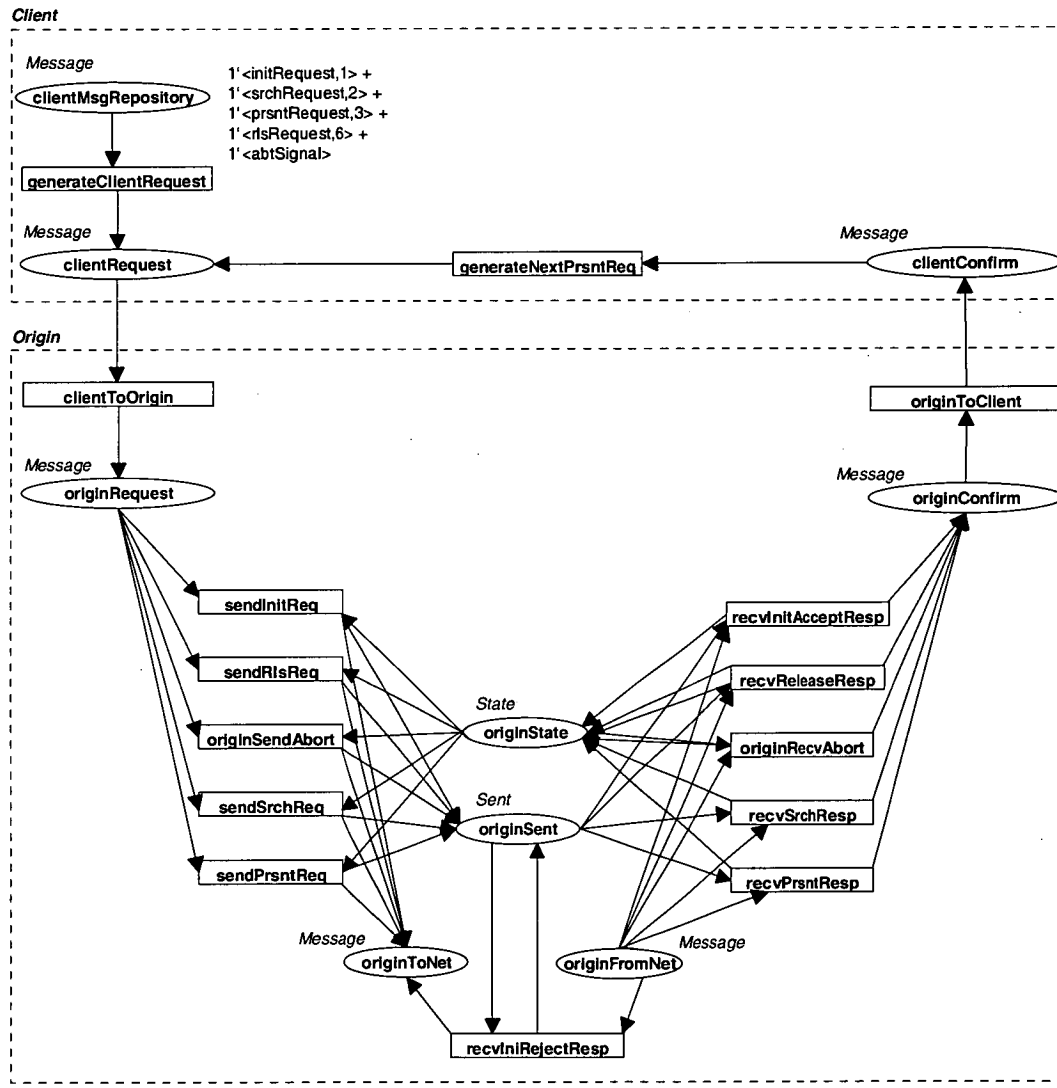


Figure 9.29: Z39.50 client

9.3.2 Performance with Segmentation

The basic Z39.50 Protocol model can be refined to include Segmentation and Access Control. Both these refinements can be achieved using subnet refinement (see Section 5.2.2). We first consider the performance of the type-subnet algorithm compared to that of the

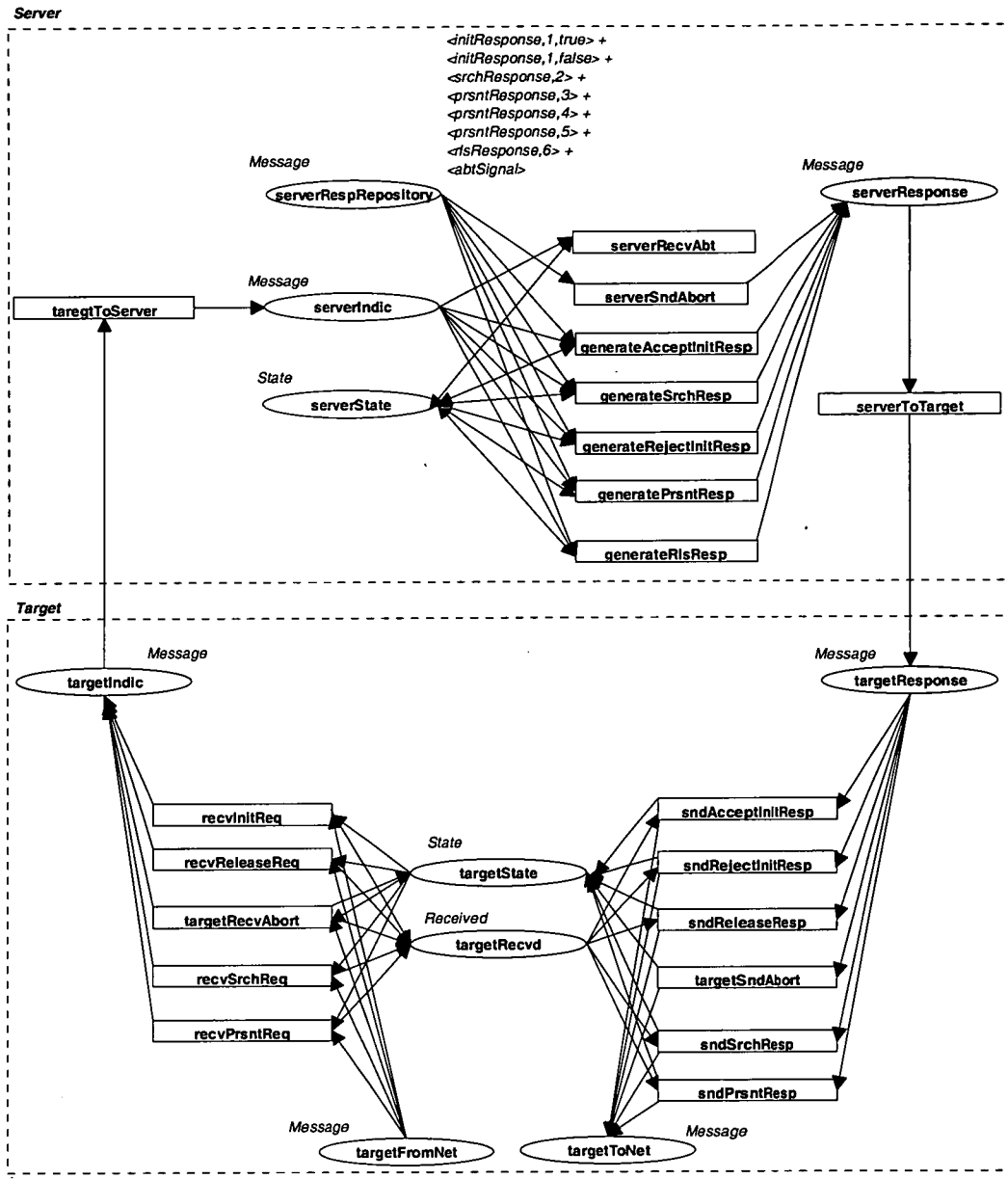


Figure 9.30: Z39.50 server

standard algorithm for the protocol refined to handle Segmentation. Segmentation is a capability added in the 1995 version of the protocol that was not present in the 1992 version (see Section 5.2.2). It allows multiple records to be returned in a single response (if the records are relatively short), and parts of records to be returned in a single response (if the records are relatively long). The subnet refinement for segmentation made to the request subnet is shown in Figure 5.7.

We have refined the model of Figures 9.29 and 9.30 to model the Z39.50 protocol with segmentation. Segmentation means that when all records will not fit in a single *Present* response the server sends a series of segments, rather than waiting for the client to request further records. To model segmentation we introduce the ability for the server to send several segments from the one *Present* request to the original server model (and constrain the *generateNextPresentRequest* transition of the client so it does not fire). The abstract

reachability graph, (i.e. the graph of the net without segmentation) has 60 057 states and 253 866 arcs and takes 144 seconds to construct, while the refined reachability graph (i.e. the reachability graph of the net with segmentation) has 320 769 states and 1 619 328 arcs. We now consider the time taken to construct the refined reachability graph using various techniques.

Originally segmentation was achieved by using subnet refinement to extend the *Message* colour to include a segment message, and by using subnet refinement to add new transitions for sending the segments.

As discussed in Section 9.2.5, the current implementation of Maria does not support mapping abstract tokens to refined tokens (in the context of markings), so extending the *Message* type effectively means the transitions connected to a place of *Message* type need to be examined for enabling using the same method as is used in the standard algorithm. A majority of places have the *Message* type and therefore when using the type-subnet algorithm most transitions needed to be examined using the same method as for the standard algorithm. Clearly this will not lead to significant performance improvement. With segmentation modelled in this manner, the standard algorithm took 1 165 seconds to construct the refined graph, and the time for the type-subnet algorithm was 1 170 seconds (plus 144 seconds to construct the abstract graph).

The type-subnet algorithm is likely to be more efficient if new places and transitions are added by subnet refinement for storing and forwarding the segment messages, instead of extending the *Message* type to include segment messages. For example, in the first approach we have the place *serverResponse* whose type is extended to handle segments, while in the second approach we have two places: the *serverResponse* place, which does not handle segment responses, and the *serverResponse_Segment* place, which does handle them. This second approach is a partial unfolding of the first. The second model has the advantage that the type-subnet algorithm does not have to re-examine transitions connected to all places of type *Message* (since the type has not changed from the abstract type). In this second implementation the standard algorithm takes 1 053 seconds to construct the refined graph, whereas the time for the type-subnet algorithm is 587 seconds (plus 144 seconds to construct the abstract graph).

9.3.3 Performance with Access Control

In this section we consider the performance of the type-subnet algorithm compared to that of the standard algorithm for the Z39.50 protocol refined to include access control. Access control introduces an access rights challenge into the middle of the normal request-response interaction. An access control challenge can be sent by the target to the origin in response to any or all of the origin requests, including *Initialize*, *Search*, and *Present*. If the origin response is acceptable then the original request operation (*Initialize*, *Search*, *Present*) proceeds as if the access challenge had never occurred.

Access control can be added to the basic Z39.50 model using subnet refinement, as shown in Figure 5.6. As with segmentation, access control can be implemented by either refining the *Message* type, or by adding places and transitions (i.e. a partial unfolding of the first implementation). The graph of Figure 9.31 shows the performance of the standard algorithm and the type-subnet algorithm for both implementations. The x-axis indicates the number of states of the Z39.50 model refined with access control. The number of states is changed by varying the number of origin requests that the target challenges for access

control. (To generate the smallest number of states shown, the target challenges the *Init* request only. For the middle number of states the *Init* and *Search* requests are challenged, and for the largest number of states the *Init*, *Search*, and *Present* requests are challenged.) The time for the type-subnet algorithm is the total time (i.e. the abstract and refined graph time). The abstract graph has 60 057 states and 253 866 arcs and takes 144 seconds to construct.

As was the case for the basic Z39.50 model refined with segmentation, the type-subnet algorithm is faster than the standard algorithm for the model where the refinement is introduced by adding places, transitions, and arcs, but slower if the refinement is achieved by extending the *Message* type. The reasons for this have been discussed in Section 9.2.5 and again in Section 9.3.2.

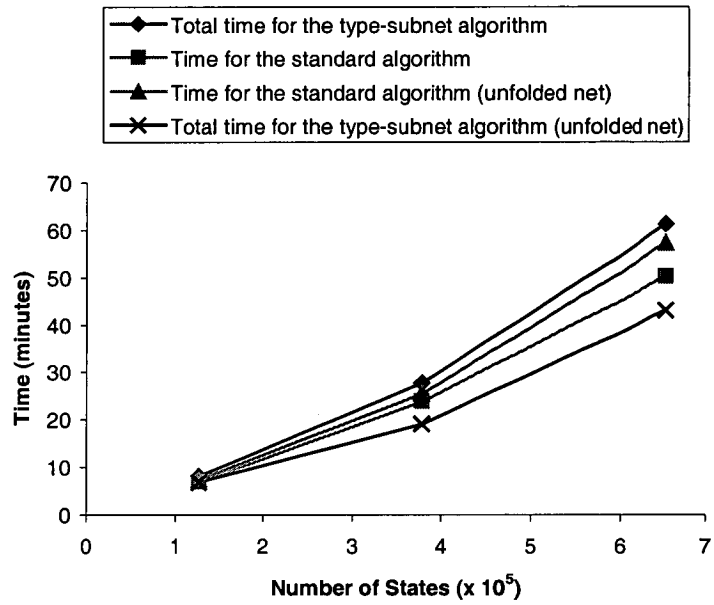


Figure 9.31: Performance of the algorithms for the Z39.50 protocol refined to include access control

9.3.4 Performance with Segmentation and Access Control

We have also implemented a Z39.50 model with both segmentation and access control. Again segmentation and access control have been implemented by extending the *Message* type, and by adding places, transitions, and arcs. With an access control challenge on each of the *Init*, *Search*, and *Present* requests, neither the standard nor type-subnet algorithm could complete due to insufficient virtual memory. With an access control challenge on only the *Init* and *Search* requests, the results were: for the implementation where the *Message* type is extended, the standard algorithm took 10 530 seconds and the type-subnet algorithm took 31 256 seconds; for the implementation where places, transitions and arcs are added, the standard algorithm took 12 162 seconds, and the type-subnet algorithm took 10 550 seconds. Once again the poor performance of the type-subnet algorithm when the *Message* type is extended is due to the fact that in the type-subnet algorithm "changed" transitions are examined using transition instance analysis (see Section 9.2.5).

9.4 Air-to-Air Missile Simulator

In Section 5.2.5 we discussed the incremental modelling of an air-to-air missile simulator. The abstract and refined models are shown in Figures 5.13 and 5.14 respectively. We have implemented the refined missile simulator model in Maria. The main problems encountered in the implementation were that real numbers are not (currently) available in Maria and functions can only be defined using macros — there is no support for iteration such as *for* and *while* loops, and there is no possibility of including external functions such as a random number generator or a square root function.

In the model presented by Gordon and Billington [82], the random number generator is used to make random changes to the target position. It simulates the target taking evasive action. Therefore, for our purposes of testing the RNSS algorithm, the following random number function has been used:

$$\text{RANDOM}(seed) = (104 * seed + 13) \bmod 81$$

This function generates a unique number in the range 0 to 80 for each seed in the same range.

The model of Gordon and Billington defines a $\text{MAG}(x,y,z)$ function. This function computes the distance from the origin to a point in 3-D space by computing $\sqrt{x^2+y^2+z^2}$. Since a square root function is not available in Maria, we approximate the $\text{MAG}(x,y,z)$ function. The square root of a number s can be found using Newton's approximation:

$$x_i = \frac{1}{2}\left(x_{i-1} + \frac{s}{x_{i-1}}\right)$$

Newton's approximation involves recursively evaluating the above function from some initial guess, x_0. Since we could not perform many iterations of Newton's approximation (Maria does not support iteration), we made an educated first guess. Our guess is based on the property that if one of x, y, or z is much larger, then it will largely determine the result:

$$x_0 = \text{MAX}(|x|,|y|,|z|) + 0.3 * \text{MEDIAN}(|x|,|y|,|z|) + 0.2 * \text{MIN}(|x|,|y|,|z|)$$

We then used an iteration of Newton's approximation:

$$\text{MAG}(x,y,z) = \frac{1}{2}\left(x_0 + \frac{x^2+y^2+z^2}{x_0}\right)$$

By testing this function for random values of x, y, and z, we found that this magnitude function gives the square root of $x^2+y^2+z^2$ to the nearest integer, which was sufficient given the constraints of integer arithmetic in Maria.

Other functions in the Missile model could be implemented as macros. A problem we encountered with this was that the Maria analyser takes a day or more just to parse the model (before reachability graph generation can even begin). Even though the functions of the Missile model are relatively complex the time taken to parse the model is rather anomalous. It appears to be caused by the expansion of the functions performed by the Maria parser. We alleviated this problem somewhat by expanding some of the functions in-line and therefore saving the parser the need to expand the functions.

Since real numbers are not supported in Maria[3] we were forced to use integers to represent the target and missile positions and velocities, as well as the system parameters (such as maximum missile velocity). Clearly the use of integer arithmetic implies that some accuracy is lost in the calculations. In order to keep this loss of accuracy to a minimum our implementation uses values representing metres and seconds rather than kilometres and hours (as in the model presented by Gordon and Billington). Even so, there is still a loss of accuracy due to the integer arithmetic, which has been observed by comparing the Maria implementation to a Design/CPN implementation by Gordon (modified to use metres and seconds and our square root approximation)[4]. This loss of accuracy means that the missile may not hit the target if the tolerance (i.e. the distance required between missile and target for a hit to be registered) is small. For the purpose of testing the performance of the RNSS algorithm compared to that of the standard algorithm it is sufficient simply to increase the tolerance. Therefore in our tests we used a tolerance of 100 metres, compared to 20 metres used in the tests presented by Gordon and Billington [82].

The number of states of the refined model depends on the number of abstract firings of the refined *Simulate* transition, which in turn depends on the initial distance between the missile and target, and the initial and maximum velocities of the missile and target. Therefore, by increasing the initial distance between the missile and target, we increase the size of the reachability graph.

We analysed the refined model using the (standard) RNSS algorithm. The graphs of Figure 9.32 and Figure 9.33 show the time performance of the RNSS algorithm and disk space used by the RNSS algorithm respectively, compared to that of the standard algorithm as the initial distance between the missile and target is increased. The x-axis of the graph indicates the value of the x-coordinate of the initial position of the target. The y and z coordinates have the value 0. The initial missile position is $(0,0,0)$ and the initial missile velocity is $(100,0,0)$. The maximum missile and target velocities are 100 and 167 ms^{-1} respectively. (Note that the missile and target velocities are maximum values, not average values. All these values were chosen so that the number of abstract firings of the refined *Simulate* transition can be easily controlled by varying the initial distance between the missile and target. This in-turn allows us to demonstrate the performance of the RNSS algorithm compared to the standard algorithm.)

[3]The Maria language is purposely limited so as to help the modeller avoid the state space explosion and to produce an efficient implementation [137].

[4]We thank Steven Gordon for providing us with a copy of this implementation.

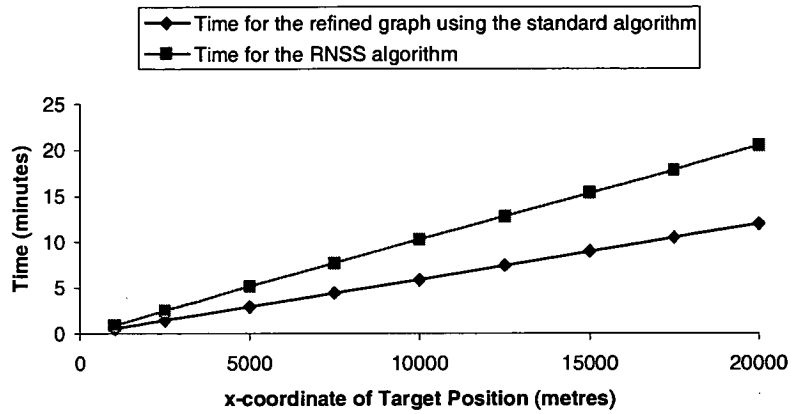


Figure 9.32: The time for the (standard) RNSS algorithm and standard algorithm as the initial distance between the target and missile is increased for the net of Figure 5.14

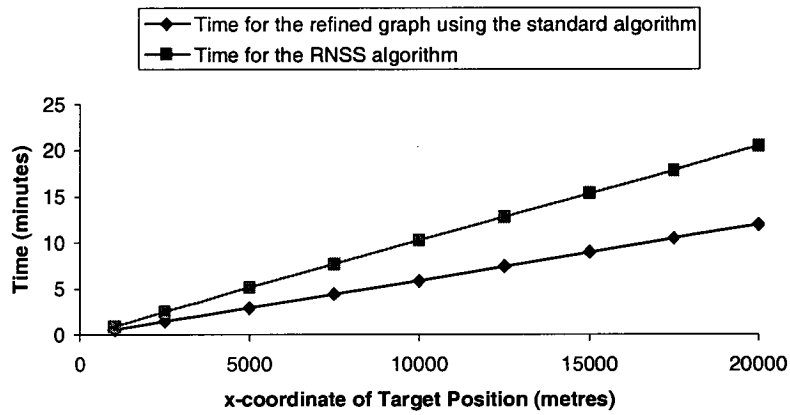


Figure 9.33: The disk space used by the full reachability graph and the (standard) RNSS as the initial distance between the target and missile is increased for the net of Figure 5.14

It can be seen in the graphs of Figures 9.32 and 9.33 that for the missile simulator the standard algorithm performs slightly better than the (standard) RNSS algorithm. This is not too surprising since a main benefit of the RNSS algorithm is to avoid the interleaving that would normally be found by the standard algorithm between transitions internal to a supernode and external transitions (see Section 9.2.8). However, using the standard algorithm for the missile simulator, we do not expect significant interleaving between the transitions of the *Simulate* supertransition, and those of the GUI. The reason being that three of the four transitions of the GUI are only enabled at the start or end of an abstract firing of the *Simulate* transition, and the other transition of the GUI (*UpdateGUI*) is only enabled half way through the abstract firing of the *Simulate* supertransition. Therefore we expect a performance penalty for the RNSS algorithm compared to the standard algorithm.

If instead the model were to have more interleaving between the GUI and the *Simulate* supertransition then the (standard) RNSS algorithm would give performance improvement compared to the standard reachability graph algorithm. We can demonstrate this by changing the *Outputs* place to have a canonical basis, as shown in Figure 9.34 (the supertransition is unchanged, and therefore is not shown in Figure 9.34). This is the model that would result from using place refinement as described in Chapter 4 on the *Outputs* place of the abstract net of Figure 5.13. Here the interleaving between *Simulate* supertransition and the remainder of the net is increased since the number of transitions in the *Outputs* superplace is increased. The results for this model as the distance between the missile and target is increased are shown in Figures 9.35 and 9.36.

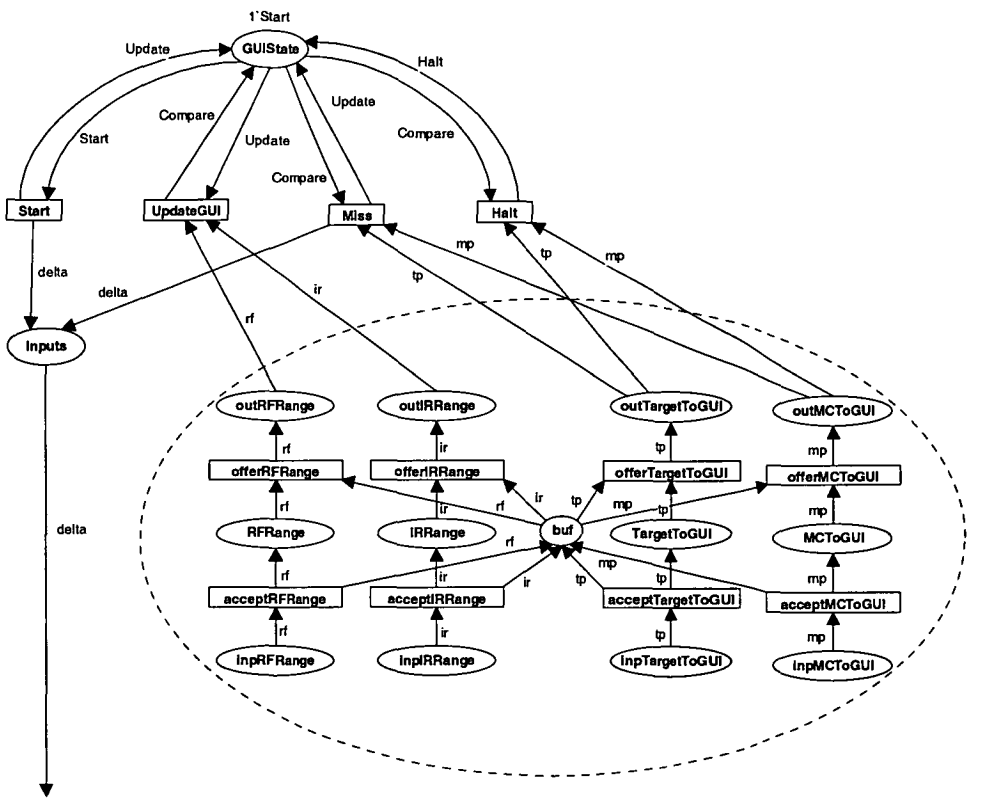


Figure 9.34: Refined missile simulator model with canonical *Outputs* superplace

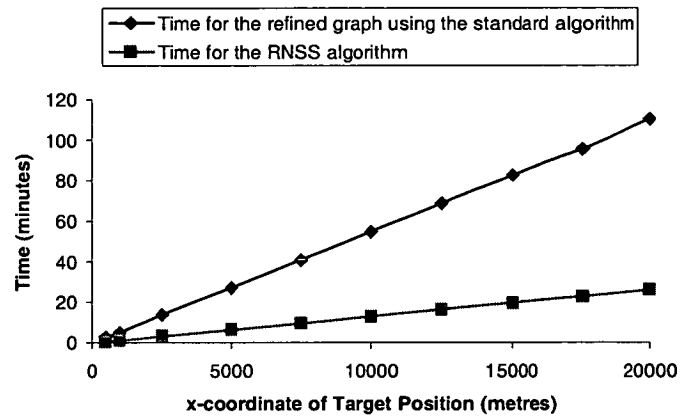


Figure 9.35: The time for the (standard) RNSS and standard algorithms as the initial distance between the target and missile is increased for the net of Figure 9.34

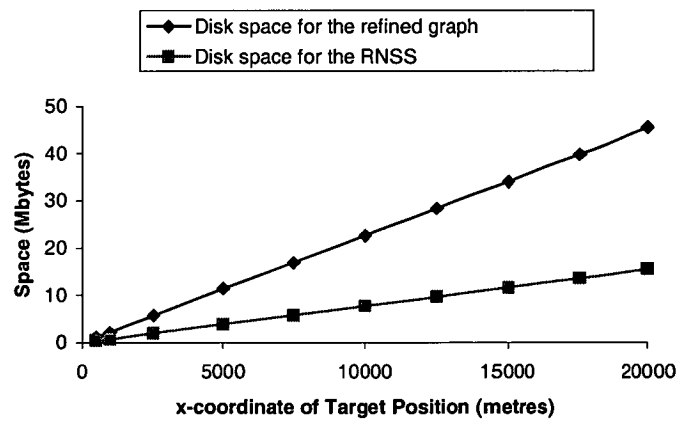


Figure 9.36: The disk space used by the full reachability graph and the (standard) RNSS as the initial distance between the target and missile is increased for the net of Figure 9.34

In the previous paragraphs we considered using the standard RNSS algorithm (Algorithm 7.5) for the missile simulator model. We now consider using the optimised RNSS algorithm presented in Section 7.4.15. As shown in Figure 9.37, for each output border transition of the *Simulate* supertransition, we add to the model a place to indicate that the border transition has fired, and we add the *finish* transition. This matches the construction required for the optimised RNSS algorithm as presented in Section 7.4.15.

In order to use the optimised algorithm (of Section 7.4.15) the *finish* transition must be *locally live*. Part of the locally live definition (Definition 7.31) requires that every internal sequence leading to the occurrence of the *finish* transition can be reordered to a sequence where the border input transitions and switch transition do not occur before the *finish* transition has occurred. In the case of the *Simulate* supertransition the input border transition (i.e. the *begin* transition) is only enabled at the start and after the occurrence of the *Miss* transition of the GUI. Since the *Miss* transition is not enabled until after all components of the *Simulate* supertransition have finished (i.e. until all output border transitions have occurred) then the *begin* and *switch* transitions cannot occur following the *switch* transition but before the *finish* transition. Thus this requirement of the locally live definition (Definition 7.31 (b) and (c)) is trivially satisfied. Further, since each component of the *Simulate* supertransition merely performs some computation, it is clear that every sequence following the *switch* transition can be extended to a sequence that involves the *finish* transition, as required by Definition 7.31 (a). (This is confirmed by the implementation, which tests for this condition.)

Thus the *finish* transition of the supertransition is locally live, and we can use the optimised RNSS algorithm to find the dead markings of the missile simulator. The time and disk space performance for the RNSS algorithm are presented in Figures 9.38 and 9.39. Here the full reachability graph could not be constructed when the initial value of the target position was 500m or more from the missile, since there was insufficient virtual memory. On the other hand, the RNSS could be constructed in less than an hour when the x-coordinate was set to 20 000m. We were not able to test values greater than approximately 20 000m since this led to integer overflow. Thus, in this case, the size and time taken to construct the RNSS is not a limiting factor.

We note that in this case the optimised RNSS algorithm performs significantly better than the standard algorithm. The reason for this is mostly due to the fact that in the standard algorithm the *finish* transition does not have to occur before a new abstract firing of the *Simulate* supertransition can begin, and is therefore enabled at the end of the first abstract firing and during all abstract firings that follow due to the occurrence of the *Miss* transition. On the other hand, in the optimisation the *finish* transition must occur before the next abstract firing can begin. This demonstrates that the optimised RNSS algorithm can produce significant performance improvements over the standard reachability algorithm.

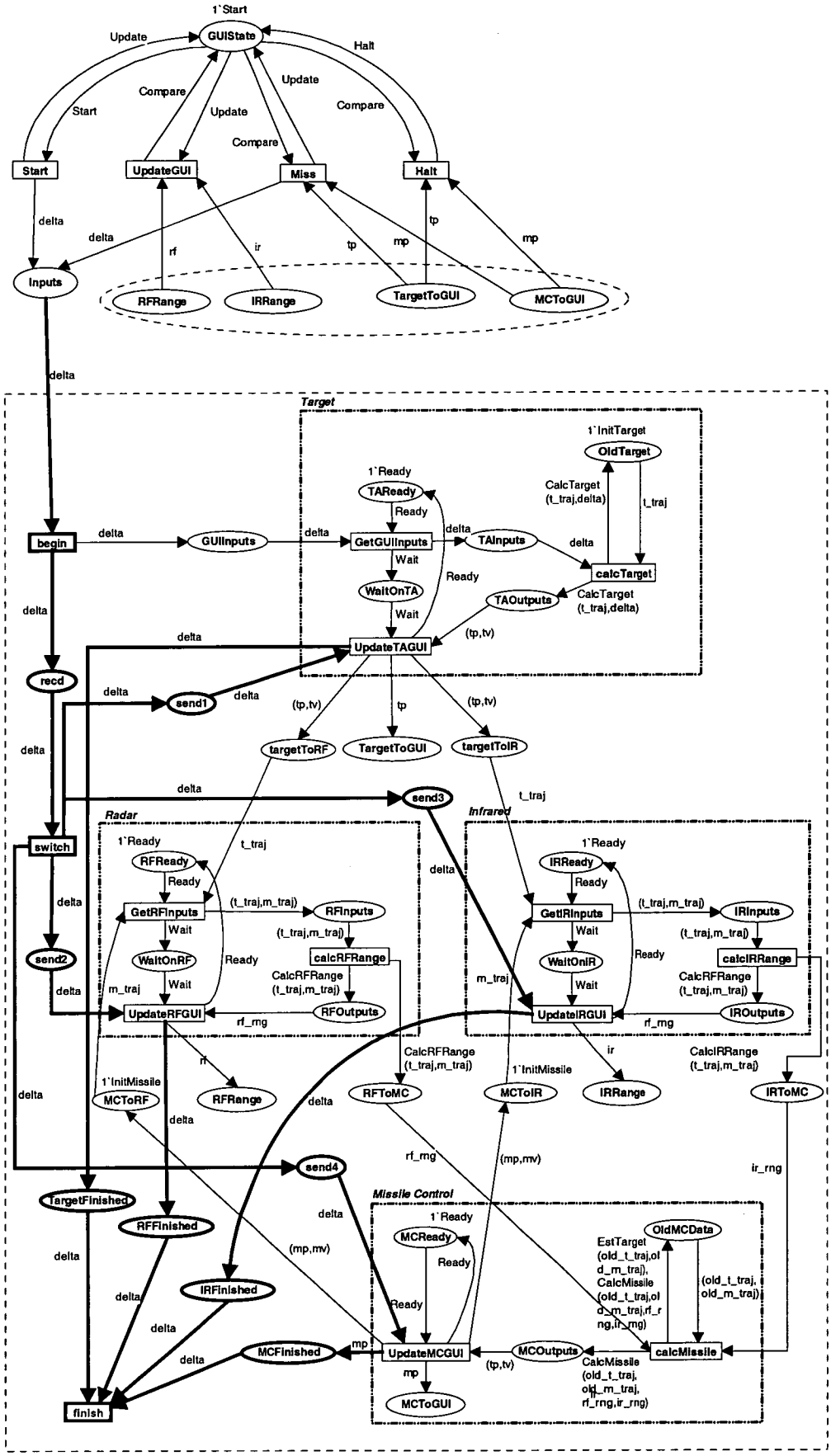


Figure 9.37: Refined missile simulator model

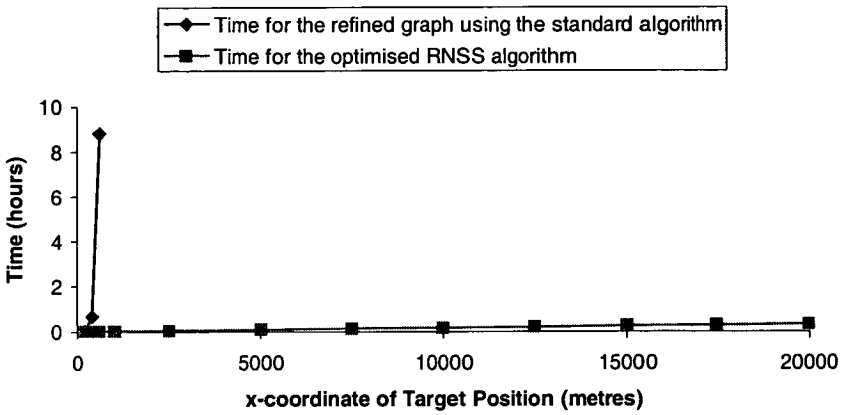


Figure 9.38: The time for the RNSS and standard algorithms as the initial distance between the target and missile is increased for the net of Figure 9.37

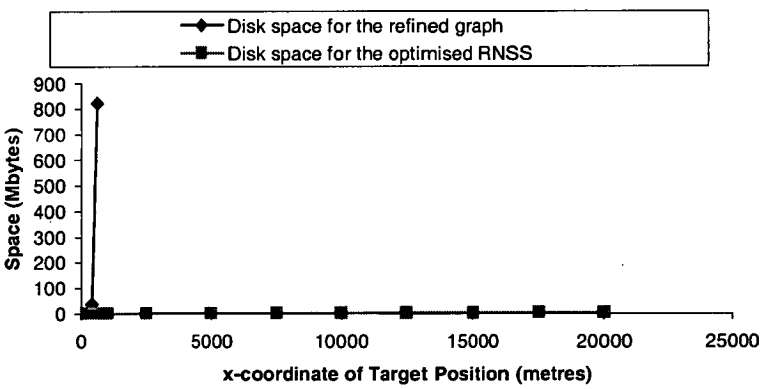


Figure 9.39: The disk space used by the full reachability graph and RNSS as the initial distance between the target and missile is increased for the net of Figure 9.37

9.5 Summary

In this chapter we have examined the performance of the incremental algorithms developed in Chapter 7. The type-subnet algorithm has been shown to give significant time improvements in situations where there are many refined firing elements that map to disabled abstract elements. The type-subnet algorithm has also been shown to perform much better than the standard algorithm when the functions associated with the arcs of the abstract graph are complex (in terms of time to compute them), and when the tokens of the abstract and refined net hold a relatively large amount of data. The RNSS algorithm has been shown to give significant time and space savings where there is interleaving between the transitions of refined nodes and transitions of other refined nodes, or external transitions

These benefits have been observed for real world studies — both the Z39.50 protocol, and the Air-to-Air Missile Simulator. For the Z39.50 protocol, the type-subnet algorithm was faster when subnet refinement was used to add places and transitions (rather than extending the type of existing places and transitions). For a large range of initial settings the optimised RNSS algorithm could be used to find the dead markings of the missile simulator in less than an hour, whereas the standard algorithm could not complete.

Chapter 10

Conclusions and Future Work

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.
T. S. ELIOT

There are several significant research results in this thesis that have arisen from investigating incremental specification and analysis. The contributions of Part I and Part II of the thesis are considered in Sections 10.1.1 and 10.1.2 respectively. Areas for future work are discussed in Section 10.2.

10.1 Contribution of the Thesis

10.1.1 Part I — Incremental Development

Incremental development involves creating a new specification or implementation by modifying an existing one. This is a commonly used technique for handling complex systems in hardware and software engineering. In fact, incremental development is fundamental to *object-orientation*, the widely adopted approach to software engineering which uses the mechanism of *inheritance*.

The value of using incremental change to achieve conceptual specialisation has been widely recognised [183, 173, 140]. In Chapter 3 we surveyed various proposals for constraining incremental change [148, 35, 7, 19, 202, 17]. An abbreviated version of this survey originally appeared in [124] and was due to both the current author and Lakos. The extensions to the survey are due to the current author. Typically, such proposals for constraining incremental change focus on the substitutability of the incrementally changed component and are based on some process algebra correctness relation, or require that a bisimulation relation hold between the original and modified components.

Concerns have been raised that constraints that require substitutability are too strong for use in practice for refinement. These concerns are supported both by our own experience, and by the incremental change used in the case studies that we examined in Chapter 5. Further to this the constraints imposed are not statically checkable and there is commonly no guide as to the changes that can be made so that the required conditions hold. We believe it is for these reasons that typically only signature compatibility is required in practice. For example, the UML (v1.3), LOOPN++ [115], Esser's OOPN language [71], and PN talk [46] do not impose constraints on the behaviour of an incrementally changed class, but only require signature compatibility that can be statically checked.

What we need, therefore, are constraints that lie between full behaviour compatibility and signature compatibility, where maximum possible expressive power is supported while maintaining conceptual specialisation. The proposal should guide the developer to the type of change that can be made, and the constraints should be statically checkable. Of utmost concern is that constraints are usable in practice. Incremental CPN Modelling, as proposed by Lakos [116], guides the developer to the forms of change possible, and is statically checkable.

Incremental CPN Modelling as proposed by Lakos [116] was presented in Chapter 4. It consists of a general constraint on incremental change — each refined behaviour must have a corresponding abstract behaviour — which is formulated in terms of morphisms on CPNs. In order to guide the developer to the forms of change that can be used to ensure this principle, Incremental CPN Modelling also provides three specific forms of refinement for CPNs: *type refinement*, *subnet refinement*, and *node refinement*. Under type refinement the structure of the net remains the same, and additional information is incorporated in the tokens and firing modes. Each value of a refined type can be projected onto a value of the abstract type. Subnet refinement involves augmenting a subnet with additional places, transitions, and arcs, or the extension of a token or mode type to include extra values which are independent of previous processing. (The values of the extended type are not projected onto values of the abstract type, but are ignored in the abstraction.) Node refinement involves replacing a place (transition) by a place (transition) bordered subnet. Canonical forms of node refinements have been proposed.

In the final part of Chapter 4 we examined the relationship between Incremental CPN Modelling and bisimulation, showing that if additionally the refined net is *at least as live as* the abstract net, then the abstract net is bisimilar to the refined net. This final part of Chapter 4 was joint work between the current author and Lakos [117].

The main contribution of Part I of this thesis has been to examine the practical applicability of Incremental CPN Modelling. In order to asses the applicability of Incremental CPN Modelling in practice the current author has examined a number of case studies [41, 179, 29, 37, 62, 112, 113, 40, 74, 73, 127, 123, 122, 143, 101, 82, 83]. Several of these studies have been presented in Chapter 5. The behavioural compatibility requirements imposed by Incremental CPN Modelling seem to match the kinds of abstraction the designers have used in the process of developing these formal specifications. Thus the majority of the incremental change observed can be achieved using Incremental CPN Modelling, whereas other proposals often would not allow the incremental change used in these case studies. Our assessment is therefore that Incremental CPN Modelling is widely applicable in practice. We believe that its use would help to clarify the models and guarantee the conceptual specialisation which is in the mind of the developer, as well as guide the developer to the appropriate forms of incremental change, and (as is shown in the sec-

ond part of this thesis) provide a mechanism to help alleviate the state space explosion problem.

10.1.2 Part II — Incremental Analysis

Formal analysis of a model is considered essential in many domains including embedded systems, safety and security critical systems, and protocol development and analysis. One of the main benefits of developing a formal model is that the model can be formally analysed.

State based methods are one of the most successful strategies for the formal analysis of concurrent systems. Unfortunately, owing to simple combinatorics, the number of states of a system increases rapidly as the complexity of the system increases. This means that the total number of states of a system is often far too large with respect to time and space resources to be fully generated, even for only a small model.

State space explosion does not preclude the use of state space analysis in practice. As we examined in Chapter 6, the great advantages of these methods have motivated many researchers to try to seek ways of alleviating the problem. These approaches to alleviating the state space explosion generally fall into one of five categories: removing information, compression, compositional techniques, preprocessing, and partial state space exploration.

The main contribution of Part II of this thesis has been the presentation of a new approach that uses the incremental change identified in Part I to help alleviate the state space explosion. Part II of the thesis is due to the current author.

The incremental algorithms were given in Chapter 7. Algorithms that cater for type, subnet, and node refinement, were presented. An algorithm that caters for a combination of type and subnet refinement was given as was an algorithm that caters for a combination of all three forms refinement. The algorithm catering for type and subnet refinement produces the full reachability graph, while the algorithm that caters for node refinement produces a set of graphs which we refer to as the *Refined Node State Space* (RNSS).

We proved that the graphs of the RNSS can be combined to give the full reachability graph, and showed how the various dynamic properties (reachability, dead markings, liveness, home properties, boundedness) can be determined directly from the RNSS without first combining the graph. (This is particularly important since combining the graphs may be an expensive operation.) We has considered an optimisation to the RNSS algorithm that can be used if the *finish* transition of each supertransition is *locally live*. In this case the unfolded optimised RNSS is not isomorphic to the full reachability graph, however, we proved that their dead states are the same. We also compared the RNSS to the modular state space of Christensen and Petrucci, highlighting situations where the RNSS algorithm will lead to considerable performance improvement over the Modular State Space algorithm.

In Chapter 8, we considered the implementation of the incremental algorithms. The Maria reachability analyser [136] was selected as the most appropriate of the available tools for the implementation of the algorithms. Maria has been modified so that the following methodology can be adopted for analysing incrementally developed models. First, the abstract and refined nets are parsed. If the incremental algorithm requires the abstract reachability graph then it is developed (if it does not already exist), or the internal representation is created in system memory (if it does exist). The refinements from the abstract

net are then detected, and the incremental algorithm is used to develop the reachability graph of the refined net.

Maria has a modular design which was ideal for this project. Modifications were made to the *Front-end Parser and Internal Representation*, *State*, *Method*, *Transition Analysis*, and *User Interface* modules in order to implement the incremental algorithms. The modifications to the *Front-end Parser* involved adding constructs to the Maria language to support the detection of the various refinements. Classes for superplaces, supertransitions, and lists of global markings were added to the internal representation generated. The *State* module was modified to support the efficient implementation of the UPDATE function, multiple reachability graphs, and SCCs.

Currently, the data structures of Maria do not provide optimum support for incremental analysis. The function EDGESFROM-TYPESUBNET from Algorithm 7.8, requires us to determine those refined firing elements which map to enabled abstract firing elements. In order to do this, we need to be able (within the context of markings) to map from refined token elements to abstract token elements and back again. While the former direction is easily supported, the latter is not. Computing this information on the fly requires the same amount of effort as the transition instance analysis algorithm of Maria. Performance improvements can therefore be expected with special support for these mappings. This is a matter for further research.

Finally, in Chapter 9 we examined the performance of the incremental algorithms. We demonstrated that one significant advantage gained by the type-subnet algorithm is that refined firing elements do not have to be considered if the corresponding abstract firing element is disabled. Therefore we can expect good performance improvement if there is a large number of refined firing elements for which the corresponding abstract firing element is disabled.

The type-subnet algorithm was also shown to perform much better than the standard algorithm when the functions associated with the arcs of the abstract graph are complex (in terms of computation time), and when the tokens of the abstract and refined net hold a relatively large amount of data.

One disadvantage of the implemented version of the type-subnet algorithm is that since the abstract graph is used to determine the enabled abstract firing modes, both the abstract and refined graphs must be represented in system memory. This means that if the abstract graph is large then the performance of the incremental algorithm may not be as good. On the other hand, if the refined graph is much larger than the abstract graph, then the amount of extra memory used to store the abstract graph becomes insignificant.

We demonstrated that for a net with refined nodes, the RNSS algorithm can save time and space since, unlike the standard graph, it will not consider all the possible interleavings of the internal activity of the refined nodes. As the extent of interleaving between internal and external transitions of the net increases, so too does the performance improvement of the RNSS algorithm compared to that of the standard algorithm.

Apart from the specially constructed examples already mentioned, we also implemented two separate case studies to assess the performance of the incremental algorithms in practice: the *Z39.50 Protocol for Information Interchange* [122] and the *Distributed Missile Simulator Model* [82]. Both of these case studies have been developed incrementally.

Subnet refinement can be used to introduce both segmentation and access control to the basic Z39.50 Protocol model. We showed that if the segmentation and access control messages are introduced by adding places and transitions then the type-subnet algorithm is faster than the standard algorithm. On the other hand, if the new messages are introduced by extending the message type, then the type-subnet algorithm is slower than the standard algorithm. The drop in performance for subnet extension can be attributed to the fact that the implementation uses the standard method to find the enabled firing modes of a transition connected to a place whose type has been changed (as discussed earlier in this section).

Node refinement is used in the Missile Simulator model to include the details of the simulation, including the radar, infrared, target, and missile control components. The main problems encountered in the implementation of the Missile Simulator were that real numbers are not available in Maria and functions can only be defined using macros. To help avoid loss in accuracy due to the use of integer arithmetic we used metres and seconds for the distances and velocities, rather than kilometres and hours.

The number of states of the refined Missile simulator model depends on the number of abstract firings of the supertransition, which in turn depends on the initial distance between the missile and target, and the initial and maximum velocities of the missile and target. Therefore, by increasing the initial distance between the missile and target, we increase the size of the reachability graph. We demonstrated that the standard RNSS algorithm gives performance improvement if there was interleaving between the supertransition and the remainder of the net. Further, we showed that the optimised RNSS could be constructed in less than an hour when the x-coordinate was set to 20 000m, whereas the full reachability graph could not even be constructed when the initial value of the x-coordinate target position was 500m.

10.2 Future Work

Several areas for further work have emerged from this thesis.

In Chapter 4 we considered the relation between Incremental CPN Modelling and bisimulation (which guarantees substitutability). We showed that if the refined net is *at least as live as* the abstract net, then the abstract net is bisimilar to the refined net. It would be worth investigating static constraints that guarantee the *at least as live as* property, since these would guarantee substitutability of the refined net, which may be desired.

In Chapter 5 we assessed the practical applicability of Incremental CPN Modelling. Our assessment is that the proposals are widely applicable in practice. However, the examination of the case studies identified that it would be advantageous to extend the existing definitions to include the notion of time. Additionally, it would be interesting to consider if the incremental algorithms can be modified to cater for, and take advantage of, timed models.

As we have seen in Chapter 6, two methods can often be combined to give the sum of reductions of the individual methods. It would be valuable to investigate the extent to which the incremental algorithms could be combined with other state space reduction techniques. The incremental approach seems to be orthogonal to the Symmetric Occurrence Graphs. Also, while it does reduce interleaving between the activity of different supernodes, it may still be improved by combination with the Stubborn Set approach.

Incremental algorithms have been presented in Chapter 7. We would like to consider whether improvements can be made to these algorithms. In particular, we would like to consider under what situations changing the abstract reachability graph *in situ* to form the refined reachability graph will lead to performance improvement. It would also be of interest to examine whether it is useful to group those places, transitions, and arcs added by subnet refinement in a module and use Modular Analysis to develop the state space.

We have shown that it is possible to determine properties of the net — reachability, dead markings, liveness, home properties, and boundedness — directly from the RNSS, without needing to unfold it. If one is prepared to sacrifice the capability of recovering the full reachability graph, then it is possible to optimise the incremental algorithm even further.

We have already considered one such optimisation. This optimisation can be applied when the *finish* transition of each supertransition is *locally live.* In this case the unfolded optimised RNSS is guaranteed to have the same dead markings as the full reachability graph.

An optimisation to the RNSS algorithm that we would like to consider is to analyse the abstract modes of a supernode separately. In the current RNSS algorithm, when finding the successors in the global graph due to a supertransition, we consider those successors resulting from step sequences that contain occurrences of the input border transitions with different abstract modes. The reason for this is that the abstract firing modes of the supernode can be dependent. For example, there may be a sequence of a supertransition that only completes if an interleaving of abstract modes is considered. If we can ensure or detect when abstract modes are independent then each abstract mode can be analysed separately. Analysing the abstract modes separately may be significantly more efficient than the current approach since the interleaving of abstract modes is not considered. Similarly, it may be possible to formulate an independence requirement for superplaces such that it is not necessary to consider all interleavings of the internal transitions of a superplace.

As discussed in Chapter 8, we would like to develop data structures for storing and accessing markings that are tailored to the special needs of incremental analysis. The data structures would provide special support for mapping between abstract token elements and refined token elements (see Section 8.5.1). In the implementation of the incremental algorithms, we modified the state storage mechanism so that given a state number, the marking of a particular place or set of places could be retrieved and changed. The modification required extra information to be stored in the state space, thus incurring some overhead. We would like to develop a state storage mechanism tailored to incremental analysis that allows direct access to place markings. Since the state space of the abstract net usually has some of the same markings as the state space of the refined net, then it may even be possible to design a data structure which would support sharing of state information between abstract and refined graphs.

In Chapter 9 we examined the performance of the incremental algorithms developed in this thesis. We demonstrated that in some situations the calculation of the SCCs of supernode reachability graphs can lead to significant state space reduction, while in other situations we pay an overhead for calculating SCCs with no benefit. We would like to develop a heuristic that can be applied to a net to determine whether it is useful to calculate SCCs. Such a heuristic would probably make use of the number of cycles in the net.

As Valmari observes alleviating state space explosion is an ongoing research challenge [194]. This thesis has contributed to this challenge by harnessing Incremental CPN

Modelling. This form of modelling has behavioural compatibility constraints which seem to be widely applicable in practice. Further, these constraints can be used by incremental analysis algorithms to reduce the effects of state space explosion. We believe that additional techniques based on the structure of models captured by the designer is a fruitful area for further research.

Appendix A

The Unified Modelling Language (UML)

It is not practical to introduce the whole UML. Good references include the UML Specification [6], The UML User Guide [33], and The UML Reference Manual [171]. Here we simply present the notation for class diagrams (Section A.1) and statechart diagrams (Section A.2).

A.1 Class Diagrams

In a class diagram, a class is represented graphically as a rectangle. Attributes of the class can be listed in a compartment below the class name, and operations can be listed in a compartment below this. For example the class diagram of Figure A.1, defines the classes `Vehicle`, `Car`, `Person`, `Company`, and `Department`. There are three main types of relationships among classes: *dependencies*, *generalisations*, and *associations*.

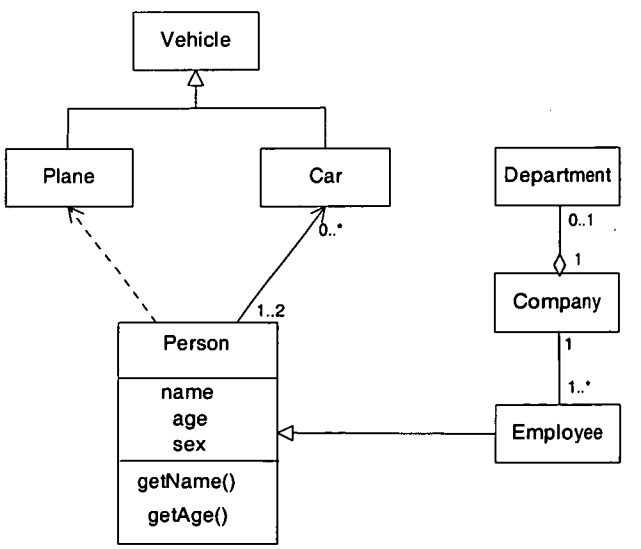


Figure A.1: A class diagram represented using the UML

A *dependency* is a relationship that states a change in specification of one class (e.g. `Plane`) may affect another class that uses it (e.g. `Person`). Graphically, a dependency is rendered as a dashed directed line.

A *generalisation* (or *specialisation*) is a relationship between a general class (e.g. `Vehicle`) and a more specific kind of that class (e.g. `Car`). Graphically, generalisation is rendered as a solid directed line with an open arrowhead, pointing at the parent.

An *association* is a relationship that specifies that objects of one class (e.g. `Employee`) are connected to objects of another class (e.g. `Company`). Graphically, an association is rendered as a solid line. The number of objects connected across an instance of an association — the multiplicity of the association — can be specified using an expression that evaluates to a value or range of values. A multiplicity at one end of an association indicates that for each object of the class at the other end, there must be that many objects at the near end. Unless otherwise specified, navigation across an association is bidirectional. Unidirection navigation can be specified by adding to the association an arrowhead pointing in the direction of traversal.

A special type of association is *aggregation*, which is a "whole/part" relationship, in which one class, the whole, (e.g. `Company`) consists of smaller classes, the parts (e.g. `Department`). Graphically, an aggregation is represented as an association with an open diamond at the whole end.

A.2 Statechart Diagrams

The dynamic behaviour of a system (often an object) can be described in the UML by using a form of state transition diagrams derived from Harrel's Statecharts [88]. A statemachine can be viewed as a directed graph where the vertices and edges represent *states* and *transitions* between the states respectively. When an object is in a state, it is generally idle, and waiting for an *event*. An *event* is any external influence on the object. A *transition* is a relationship between two states indicating that when a specified event occurs, and the specified guard conditions of the transition are satisfied, then the object will perform certain *actions* and enter the second state. An *action* is an executable atomic computation. Actions may include operation calls (to the object that owns the state machine, as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object.

Instead of being idle whilst waiting for an event, a state may perform an *activity*. An activity is an ongoing non-atomic execution. Also, a transition may be triggerless (that is, a transition with no event trigger). In this case the transition is triggered implicitly when its source state has completed its activity. Figure A.2 is an example of a UML state machine for a home thermostat.

States of a UML state machine may have disjoint or concurrent sub-states. Such states are known as *complex states*. Similarly, transitions may have several concurrent sub-states as a source or target. Such transitions are known as *complex transitions*. The use of concurrency is constrained for practical reasons [6, p. 2-157]. To avoid "inconsistencies, including deadlocks, multiple occupation of a state and other problems" [171, p. 211], the UML requires a complex state be able to be decomposed into an *and-or-tree* [171, p. 211], where *and* (*or*) nodes correspond to a state with concurrent (sequential) sub-states. A simple transition must connect two states in the same sequential region, or two

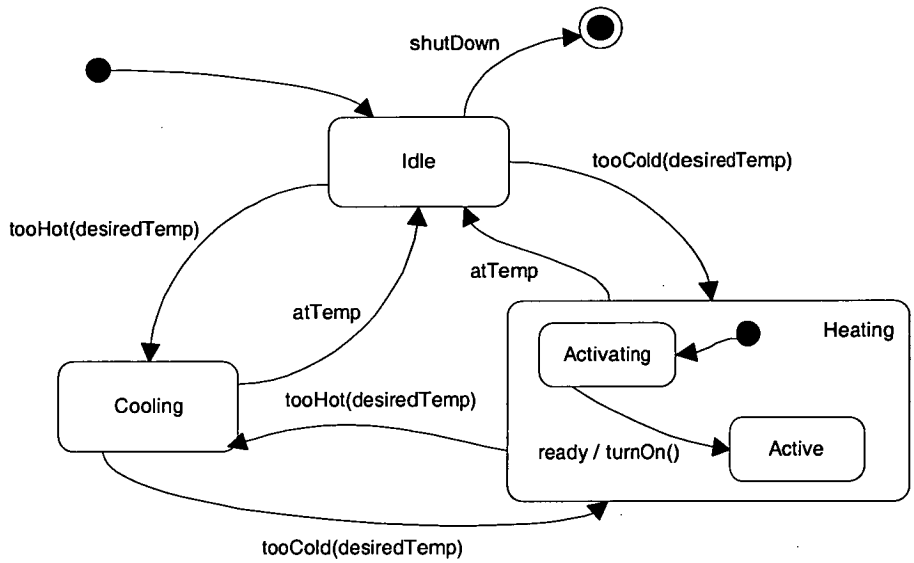


Figure A.2: A UML state machine of a home thermostat [33, fig. 21-1]

states separated by 'or' levels only. A complex transition entering (leaving) a concurrent region must enter (leave) each subregion of that concurrent region. A complex transition is represented using a bar notation. A typical example of a concurrent composite state with complex transitions entering and leaving it is given in Figure A.3.

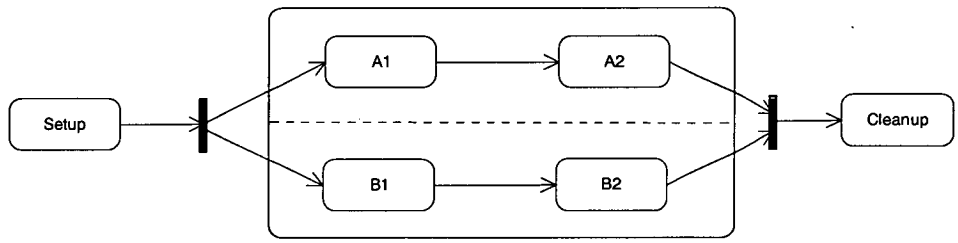


Figure A.3: A complex state [171, fig. 13-55]

Appendix B

Tarjan's Algorithm to compute SCCs

In Algorithm B.1 we present Tarjan's algorithm. In order to simplify the description, the notation we use differs from the original presentation [184] but is similar to that of Nuutila [149].

The algorithm COMPUTESCCS($\mathcal{G}$) finds the strongly connected components of a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. This algorithm repeatedly calls the recursive procedure VISIT($\mathcal{G}, v, stack$). Each vertex v has associated variables *inComponent* and *visited*. We denote the variable associated with a node $v \in V$ using a dot notation (*v.visited* and *v.inComponent*). The VISIT function enters the nodes of the graph in a depth-first order. For each strongly connected component C, the first node of C entered is called the *root* component of C. The variable *v.root* contains a candidate node of the root of the component containing v, and the main goal of the algorithm is to find the component roots. In this algorithm, the node v is the initial root candidate and new root candidates are obtained by processing the edges that leave v. The MIN(v, w) function compares v and w with respect to the order in which VISIT has entered them. When all edges leaving v have been processed, *v.root* is equal to v (*v.root* $= v$) if and only if v is the root of the component containing v.

The variable *v.inComponent*, which is initially *false*, distinguishes between nodes belonging to the same component as node v and nodes belonging to other components. When a SCC is fully detected, the nodes belonging to it are on the top of the stack. They are removed from the stack and their *inComponent* variable set to *true*.

To implement the MIN function, we store a map of the vertex (state) number and a depth-first number indicating the order in which the state was visited. To avoid wasting space, we also use the depth-first number to store the *inComponent* value of the state. That is we use a single computer word to store both values. A bit-mask is used to store or retrieve these values from the given word.

Algorithm B.1 Tarjan's SCC algorithm

```
VISIT(G, v, stack) begin
  v.root := v
  v.inComponent := false
  PUSH(v, stack)
  for all (v, w) ∈ E do
    if not VISITED(w) then
      VISIT(G, w, stack)
    end if
    if not w.inComponent then
      v.root := MIN(v.root, w.root)
    end if
  end for
  if v.root = v then
    repeat
      w := POP(stack)
      w.inComponent := true
    until w = v
  end if
end

COMPUTESCCS(G) begin
  for all v ∈ V do
    v.visited := false
    v.inComponent := false
  end for
  stack := ∅
  for all v ∈ V do
    if not v.visited then
      VISIT(G, v, stack)
    end if
  end for
end
```

Appendix C

Results

In this appendix we provide the raw data from maria for the graphs presented in the thesis.

Number of database managers	Disk space for standard algorithm when storing offsets for each place (bytes)	Disk space for standard algorithm when not storing offsets for each place (bytes)	Percentage difference (%)
6	350164	303476	13.33
7	1531204	1367876	10.67
8	6508888	5948984	8.60
9	27163004	25273404	6.96
10	108650716	102352124	5.80
11	426098244	405312964	4.88

Table C.1: Disk space overhead of storing place offsets for each marking of the Database Managers Net (graphed in Figure 8.10)

Number of database managers	Time for standard algorithm when storing offsets for each place (secs)	Time for standard algorithm when not storing offsets for each place (secs)	Percentage difference (%)
6	3	3	0.00
7	15	13	13.33
8	75	64	14.67
9	350	298	14.86
10	1597	1349	15.53
11	7016	5943	15.29

Table C.2: Time overhead of storing place offsets for each marking of the Database Managers Net (graphed in Figure 8.11)

Number of transitions in sequnece	Time to construct the RNSS when the SCCs are calculated (secs)	Time to construct the RNSS when the SCCs are not calculated (secs)	Percentage Difference (%)
0	0	0	0.00
2	1	5	80.00
4	4	30	86.67
6	6	115	94.78
8	11	324	96.60
10	17	735	97.69

Table C.3: Time improvement when computing SCCs for the net of Figure 9.1 (graphed in Figure 9.2)

Number of transitions in sequnece	Disk space used by the RNSS when the SCCs are calculated (bytes)	Disk space used by the RNSS when the SCCs are not calculated (bytes)	Percentage Difference (%)
0	53444	53380	0.00
2	208948	906228	76.94
4	509604	5223428	90.24
6	956084	18137812	94.73
8	1548388	47269860	96.72
10	2286516	102728372	97.77

Table C.4: Space improvement of computing SCCs for the net of Figure 9.1 (graphed in Figure 9.3)

Number of transitions in sequnece	Time for the RNSS algorithm when the SCCs are calculated (secs)	Time for the RNSS algorithm when the SCCs are not calculated (secs)	Percentage Difference (%)
0	0	0	0.00
2	4	3	25.00
4	17	14	17.65
6	51	41	19.61
8	128	104	18.75
10	272	222	18.38

Table C.5: Time for computing SCCs for the net of Table 9.1 (without t_{n+1}) (graphed in Figure 9.4)

Number of transitions in sequnece	Disk space used by the RNSS when the SCCs are calculated (bytes)	Disk space used by the RNSS when the SCCs are not calculated (bytes)	Percentage Difference (%)
0	53444	53380	0.12
2	584188	583788	0.07
4	2467764	2466740	0.04
6	7181612	7179676	0.03
8	16825252	16822116	0.02
10	34120284	34115660	0.01

Table C.6: Space overhead for computing SCCs for the net of Table 9.1 (without t_{n+1}) (graphed in Figure 9.5)

Number of States refined graph	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
3679	30	34	-13.33
98125	600	211	64.83
273125	1604	539	66.40
528125	3054	1035	66.11
863125	4976	1721	65.41
1278125	7349	2579	64.91
1773125	10207	3632	64.42
2.348125	13643	4965	63.61
3003125	17662	6672	62.22
3894725	23831	9851	58.66

Table C.7: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of states of the refined graph increases (graphed in Figure 9.7)

Number tokens in p1 - p10	Time for the abstract graph using the standard algorithm (secs)	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Improvement (%)
1	0	0	0	0
2	5	73	71	2.74
3	7	122	99	18.85
4	9	156	105	32.69
5	12	207	120	42.03
6	15	270	142	47.41
7	18	348	168	51.72
8	22	423	186	56.03
9	26	506	199	60.67
10	30	600	211	64.83
11	35	705	239	66.10
12	39	821	267	67.48
13	45	939	275	70.71
14	50	1067	304	71.51
15	56	1208	332	72.52

Table C.8: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of disabled firing elements is increased (graphed in Figure 9.8)

Value of *max*	Time for the abstract graph using the standard algorithm (secs)	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
0	9	156	105	32.69
100	18	217	113	47.93
200	27	276	119	56.88
300	35	334	131	60.78
400	45	393	140	64.38
500	53	452	148	67.26
600	62	512	157	69.34
700	71	570	166	70.88
800	79	629	175	72.18
900	88	688	183	73.40
1000	96	746	192	74.26

Table C.9: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the complexity of the arc functions is increased (graphed in Figure 9.10)

Avaliable RAM (Mbytes)	Time for the refined graph using the standard algorithm (secs)	Time for the abstract graph using the standard algorithm (secs)	Total time for type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
256	413	260	345	16.46
206	413	260	344	16.71
156	413	260	344	16.71
106	413	259	344	16.71
56	412	259	348	15.53
45	414	260	351	15.22
41	414	260	354	14.49
36	415	260	405	2.41
26	415	260	406	2.17
16	416	263	402	3.37
6	416	263	411	1.20
0	416	262	402	3.37

Table C.10: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of available RAM is increased (graphed in Figure 9.11)

Number refined places	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
0	157	105	33.12
1	157	118	24.84
2	157	118	24.84
3	157	126	19.75
4	158	128	18.99
5	158	137	13.29
6	159	138	13.21
7	159	147	7.55
8	160	148	7.50
9	160	157	1.88
10	160	158	1.25
11	160	157	1.88
12	160	158	1.25
13	160	157	1.88
14	160	157	1.88
15	160	157	1.88

Table C.11: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the number of refined places in the net of Figure 9.6 (b) is increased (graphed in Figure 9.12)

Number strings in tokens	Time for the abstract graph using the standard algorithm (secs)	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
0	9	156	105	32.69
1	27	382	242	36.65
2	46	610	383	37.21
3	64	846	522	38.30
4	84	1097	672	38.74
5	104	1369	820	40.10
6	124	1651	984	40.40
7	146	1953	1146	41.32
8	168	2264	1317	41.83
9	188	2585	1480	42.75
10	209	2936	1676	42.92

Table C.12: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of data in non-refined tokens is increased (graphed in Figure 9.15)

Number strings in refined structure	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
0	156	105	32.69
1	175	125	28.57
2	194	147	24.23
3	214	167	21.96
4	237	192	18.99
5	258	213	17.44
6	282	241	14.54
7	304	263	13.49
8	328	289	11.89
9	354	316	10.73
10	378	343	9.26

Table C.13: The performance of the type-subnet algorithm on the nets of Figure 9.6 as the amount of data in refined tokens is increased (graphed in Figure 9.16)

Number of transitions added	Time for the refined graph using the standard algorithm (secs)	Total time for the type-subnet algorithm (abstract + refined graph) (secs)	Percentage Difference (%)
0	132	58	56.06
5	178	105	41.01
10	223	151	32.29
15	270	195	27.78
20	315	241	23.49
25	360	286	20.56
30	406	332	18.23
35	451	378	16.19
40	498	423	15.06
45	542	469	13.47
50	589	514	12.73

Table C.14: The performance of the type-subnet algorithm on the nets of Figure 9.6 (a) and Figure 9.17 as transitions are added to the net of Figure 9.17 (graphed in Figure 9.18)

Number of transitions in sequence	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
2	55	0	100
4	237	1	99.58
6	714	4	99.44
8	1777	11	99.38
10	6649	25	99.62
12	19897	49	99.75
14	Insufficient memory	92	
16	Insufficient memory	162	
18	Insufficient memory	275	
20	Insufficient memory	444	

Table C.15: The time performance of the RNSS for the superplace example (Figure 9.19) (graphed in Figure 9.21)

Number of transitions in sequence	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
1	129	9	93.02
2	447	21	95.30
3	1201	48	96.00
4	4070	100	97.54
5	11575	183	98.42
6	30051	314	98.96
7	Insufficient memory	510	
8	Insufficient memory	782	
9	Insufficient memory	1164	
10	Insufficient memory	1699	

Table C.16: The time performance of the RNSS for the supertransition example (Figure 9.20) (graphed in Figure C.16)

Number of transitions in sequence	Disk space for refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
2	16578040	135612	99.18
4	68753704	547692	99.20
6	199349464	1616412	99.19
8	468043528	3877836	99.17
10	955747768	8079996	99.15
12	1767096040	15213612	99.14
14	Insufficient memory	26542812	
16	Insufficient memory	43635852	
18	Insufficient memory	68395836	
20	Insufficient memory	103091436	

Table C.17: The disk space performance of the RNSS for the superplace example (Figure 9.19) (graphed in Figure 9.23)

Number of transitions in sequence	Disk space for refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
1	36951592	1039475	97.19
2	124969000	1760243	98.59
3	319680040	3392627	98.94
4	687522856	6531059	99.05
5	1313432104	11948147	99.09
6	2302083112	20610035	99.10
7	Insufficient memory	33691763	
8	Insufficient memory	52592627	
9	Insufficient memory	78951539	
10	Insufficient memory	114662387	

Table C.18: The disk space performance of the RNSS for the supertransition example (Figure 9.20) (graphed in Figure 9.24)

Number of superplaces	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
1	0	0	0
2	8	0	100
3	2146	0	100
4	Insufficient memory	0	
5	Insufficient memory	0	
6	Insufficient memory	0	
7	Insufficient memory	0	
8	Insufficient memory	0	
9	Insufficient memory	0	
10	Insufficient memory	1	

Table C.19: The time performance of the RNSS for a net with several copies of the superplace of Figure 9.19 (graphed in Figure 9.25)

Number of superplaces	Disk space for the refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
1	20240	20352	-0.55
2	2488040	40592	98.37
3	464600040	60832	99.99
4	Insufficient memory	81072	
5	Insufficient memory	101312	
6	Insufficient memory	121552	
7	Insufficient memory	141792	
8	Insufficient memory	162032	
9	Insufficient memory	182272	
10	Insufficient memory	202512	

Table C.20: The disk space used by the RNSS for a net with several copies of the superplace of Figure 9.19 (graphed in Figure 9.26)

Number of supertransitions	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
1	2	3	-50.00
2	38	7	81.58
3	195	9	95.38
4	657	13	98.02
5	2065	16	99.23
6	9841	19	99.81
7	39910	23	99.94
8	Insufficient memory	26	
9	Insufficient memory	30	
10	Insufficient memory	33	
11	Insufficient memory	37	
12	Insufficient memory	40	
13	Insufficient memory	44	
14	Insufficient memory	47	
15	Insufficient memory	51	
16	Insufficient memory	54	
17	Insufficient memory	58	
18	Insufficient memory	62	
19	Insufficient memory	66	
20	Insufficient memory	69	

Table C.21: The time performance of the RNSS for a net with several copies of the supertransition of Figure 9.20 (graphed in Figure 9.27)

Number of supertransitions	Disk space for the refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
1	1007656	1014899	-0.72
2	11810216	2028862	82.82
3	56960040	3042825	94.66
4	183071656	4056788	97.78
5	465043496	5070751	98.91
6	1012278696	6084714	99.40
7	1974905896	7098677	99.64
8	Insufficient memory	8112640	
9	Insufficient memory	9126603	
10	Insufficient memory	10140566	
11	Insufficient memory	11154529	
12	Insufficient memory	12168492	
13	Insufficient memory	13182455	
14	Insufficient memory	14196418	
15	Insufficient memory	15210381	
16	Insufficient memory	16224344	
17	Insufficient memory	17238307	
18	Insufficient memory	18252270	
19	Insufficient memory	19266233	
20	Insufficient memory	20280196	

Table C.22: The disk space used by the RNSS for a net with several copies of the supertransition of Figure 9.20 (graphed in Figure 9.28)

Number of States of Refined Graph	Total time for the type-subnet algoithm (secs)	Time for the standard algorithm (secs)	Time for the standard algorithm (unfolded net) (secs)	Total time for the type-subnet algorithm (unfolded net) (secs)
128133	487	416	447	410
378423	1666	1431	1528	1147
655389	3684	3022	3444	2591

Table C.23: Performance of the algorithms for the Z39.50 protocol refined to include access control (graphed in Figure 9.31)

x-coordinate of target position (meters)	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
1000	33	56	41.07
2500	90	153	41.18
5000	178	311	42.77
7500	268	461	41.87
10000	354	616	42.53
12500	447	767	41.72
15000	539	920	41.41
17500	629	1068	41.10
20000	717	1228	41.61

Table C.24: The time for the (standard) RNSS algorithm and standard algorithm as the initial distance between the target and missile is increased for the net of Figure 5.14 (graphed in Figure 9.32)

x-coordinate of target position (meters)	Disk space for the refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
1000	197698	230621	14.28
2500	548290	639517	14.26
5000	1096090	1278417	14.26
7500	1621978	1891761	14.26
10000	2169778	2530661	14.26
12500	2717578	3169561	14.26
15000	3265378	3808461	14.26
17500	3813178	4447361	14.26
20000	4360978	5086261	14.26

Table C.25: The disk space used by the full reachability graph and the (standard) RNSS as the initial distance between the target and missile is increased for the net of Figure 5.14 (graphed in Figure 9.33)

x-coordinate of target position (meters)	Time for the refined graph using the standard algorithm (secs)	Time for the RNSS algorithm (secs)	Percentage Difference (%)
500	166	40	75.90
1000	298	72	75.84
2500	836	198	76.32
5000	1636	390	76.16
7500	2462	581	76.40
10000	3292	783	76.22
12500	4127	989	76.04
15000	4952	1182	76.13
17500	5729	1372	76.05
20000	6625	1570	76.30

Table C.26: The time for the (standard) RNSS and standard algorithms as the initial distance between the target and missile is increased for the net of Figure 9.34 (graphed in Figure 9.35)

x-coordinate of target position (meters)	Disk space for the refined graph (bytes)	Disk space for the RNSS (bytes)	Percentage Difference (%)
500	1199282	409183	65.88
1000	2158258	736527	65.87
2500	5994162	2045903	65.87
5000	11987762	4091319	65.87
7500	17741618	6055383	65.87
10000	23735218	8100799	65.87
12500	29728818	10146699	65.87
15000	35722418	12192115	65.87
17500	41716018	14238015	65.87
20000	47709618	16283431	65.87

Table C.27: The disk space used by the full reachability graph and the (standard) RNSS as the initial distance between the target and missile is increased for the net of Figure 9.34 (graphed in Figure 9.36)

x-coordinate of target position (meters)	Time for the refined graph using the standard algorithm (secs)	Time for the optimised RNSS algorithm (secs)	Percentage Difference (%)
200	4	6	-50
400	2392	24	99.00
600	31749	29	99.91
1000	Insufficient memory	55	
2500	Insufficient memory	149	
5000	Insufficient memory	300	
7500	Insufficient memory	443	
10000	Insufficient memory	586	
12500	Insufficient memory	743	
15000	Insufficient memory	886	
17500	Insufficient memory	1047	
20000	Insufficient memory	1177	

Table C.28: The time for the RNSS and standard algorithms as the initial distance between the target and missile is increased for the net of Figure 9.37 (graphed in Figure 9.38)

x-coordinate of target position (meters)	Disk space for the refined graph (bytes)	Disk space for the optimised RNSS (bytes)	Percentage Difference (%)
200	26378	23472	11.02
400	39538946	78574	99.80
600	862565186	113816	99.99
1000	Insufficient memory	205616	
2500	Insufficient memory	569904	
5000	Insufficient memory	1139104	
7500	Insufficient memory	1685536	
10000	Insufficient memory	2254736	
12500	Insufficient memory	2823936	
15000	Insufficient memory	3393136	
17500	Insufficient memory	3962336	
20000	Insufficient memory	4531536	

Table C.29: The disk space used by the full reachability graph and RNSS as the initial distance between the target and missile is increased for the net of Figure 9.37 (graphed in Figure 9.39)

Appendix D

Publications

D.1 Conference Papers

1. C.A. Lakos and G.A. Lewis, Behaviour Inheritance for Object Lifecycles, *33rd International Conference on Technology of Object-Oriented Languages and Systems - Europe TOOLS33*, Mont Saint-Michel, France, IEEE Computer Society Press, 2000
   Abstract The rules for inheritance of classes with respect to data and function members are well defined. For example, the proposals for programming by contract in Eiffel ensure additional consistency between superclasses and subclasses. In object-oriented design, it is common to capture the behaviour of classes with lifecycles which are expressed in the form of finite state machines. In this context, there are very few proposals for what constitutes consistency between superclasses and subclasses. This paper presents proposals for consistency between superclasses and subclasses in the context of the Petri Net formalism, which is a form of finite state machine with explicit provisions for concurrency. The paper cites the applicability of these proposals in the context of protocols, and argues for a similar applicability in the context of object lifecycles.

2. C.A. Lakos and G.A. Lewis, A Practical Approach to Incremental Specification, *IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall, 2000.
   Abstract The object-oriented specification of concurrent and distributed systems has tended to emphasise the aspect of substitutability at the expense of code reuse. A variety of constraints has been imposed in order to guarantee substitutability in one form or another. This paper argues that the incremental development of software specifications needs to consider substitutability in the context of code reuse. Further, the common approach of starting with an abstract specification and then progressively refining it (in some general way) means that many existing substitutability constraints are too strong. In the context of Coloured Petri Nets, we advocate the use of three specific forms of refinement — *type refinement*, *subnet refinement*, and *node refinement*. These have weaker demands for substitutability, namely that every (complete) refined behaviour has a corresponding abstract behaviour, but not necessarily vice versa. An examination of case studies in the literature suggests that this approach is applicable in practice.

3. G.A. Lewis and C.A. Lakos, Incremental State Space Construction for Coloured Petri Nets, *22nd International Conference on Application and Theory of Petri Nets*, (ICATPN'2001), Newcastle upon Tyne, United Kingdom, June 25-29, 2001, LNCS

2075, pages 263 – 282.

Abstract State space analysis is a popular formal reasoning technique. However, it is subject to the crippling problem of state space explosion, where its application to real world models leads to unmanageably large state spaces. In this paper we present algorithms which attempt to alleviate the state space explosion problem by taking advantage of the common practice of incremental development, i.e. where the designer starts with an abstract model of the system and progressively refines it. The performance of the incremental algorithm is compared to that of the standard algorithm for some case studies, and situations under which the performance improvement can be expected are identified.

D.2 Workshop Papers and Reports

1. C.A. Lakos and G.A. Lewis, A Practical Approach to Behavioural Inheritance in the Context of Coloured Petri Nets (Extended Abstract), *Workshop on Semantics of Objects As Processes*, Lisbon, Portugal, May 1999, BRICS Note Series, NS-99-2, H. Hüttel and J. Kleist and U. Netsmann and A. Ravara (editors), pages 21-28.

2. C.A. Lakos and G.A. Lewis, A Catalogue of Incremental Changes for Coloured Petri Nets, *Department of Computer Science, University of Adelaide*, TR99-02, 1999.
   Abstract This paper presents three forms of incremental change or refinement which are considered appropriate for Coloured Petri Nets. The intention is to recommend forms which are appropriate to Petri Nets and not primarily driven by the desire to emulate object-oriented programming languages. Nevertheless, the proposals are compared with others in the literature — with object-oriented programming languages, with practical case studies of the application of formal methods, and with other object-oriented Petri Net formalisms.

3. G.A. Lewis and C.A. Lakos, Towards Incremental Analysis, *Workshop on Formal Methods for Dependable Systems (FMDS)*, Brisbane, Australia 1998.
   Abstract As the size of a formal model increases, state space size becomes more complex in terms of time, or space, or both. This complexity means that state space analysis of a formal model is often practically impossible, even for a modest sized system. Recently, techniques to reduce the complexity have done so by taking advantage of the structure built into the model by the designer. In a similar vein we plan to take advantage of the incremental specification that is found in many formal models. This paper examines a number of case studies to determine the various types of incremental change that are used in practice, and presents a sketch of an incremental state space generation algorithm.

4. G.A. Lewis and C.A. Lakos, Incremental Reachability Algorithms, *Department of Electrical Engineering and Computer Science, University of Tasmania* TR99-01, 1999.
   Abstract This technical report records work in progress towards the implementation of reachability analysis algorithms that take advantage of the incremental specification of a system. It therefore, by way of a requirements chapter, details what is required to implement such algorithms in the context of an existing CPN analysis tool. In the introduction, we clarify what is meant by incremental reachability analysis, and why we believe that it may offer performance improvements. In section 2, we present the formal definitions of the kinds of incremental specification that appear to us to be well-behaved, and therefore amenable to incremental analysis. It is possible to get the main ideas without having a thorough understanding of the formalism. In section 3, we present the reachability analysis algorithms at

a reasonably abstract level and in section 4 we comment on the performance of these algorithms. In section 5, we isolate the main requirements for implementing the algorithms in the context of an existing analysis tool.

References

[1] Basic Reference Model for Open Distributed Processing. Iso/iec cd 10746, 1994.

[2] *CCITT Specification and Description Language (SDL).* Recommendation Z.100. International Telecommunication Union, Geneva, Switzerland, October 1996.

[3] High-level Petri Nets – Concepts, Definitions and Graphical Notation. Committee Draft ISO/IEC 15909 Version 3.4, 1997.

[4] Tina architecture. Technical Report Version 5.0, 1997. ⟨http://www.tinac.com⟩.

[5] *Information Technology—Programming Languages—C++.* ISO/IEC 14882. International Organization for Standardization, Geneva, Switzerland, 1998.

[6] *OMG Unified Modeling Langauge Specification (Version 1.3).* OMG, 1999. ⟨http://www.rational.com/uml/resources/documentation⟩.

[7] W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-Net-Based Approach. In P. Azéma and G. Balbo, editors, *18th International Conference on Application and Theory of Petri Nets (ICATPN'97), Tolouse, France, June 23-27, 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.

[8] R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[9] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *1st European Conference on Object-Oriented Programming 1987 (ECOOP'87), Paris, France*, volume 276 of *Lecture Notes in Computer Science*, pages 234–242. Springer-Verlag, 1987.

[10] P. America. Issues in the Design of a Parallel Object-Oriented Language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[11] P. America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[12] ANSI/NISO. *Z39.50-1992 Information Retrieval Service and Protocol.* 1992.

[13] ANSI/NISO. *Z39.50-1995 (Versions 2 and 3) Information Retrieval: Application Service Definition and Protocol Specification.* 1995.

[14] G.H. Archinoff, R.J. Hohendorf, A. Wassyng, B. Quigley, and M.R. Borsch. Verification of the Shutdown System Software at the Darlington Nuclear Generating Station. In *International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, UK, May 1990.

[15] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, Reading, 1996.

[16] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages. *Journal of the ACM*, 40(3):653–682, July 1993.

[17] C. Balzarotti, F. DeCindio, and L. Pomello. Observation Equivalences for the Semantics of Inheritance. In *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18 1999*, pages 67–82. Chapman and Hall, 1999.

[18] L.M. Barroca and J.A. McDermid. Formal methods: Use and Relevance for the Development of Safety-Critical Systems. *The Computer Journal*, 35(6):579–599, December 1992.

[19] T. Basten. *In Terms of Nets: Systems Design with Petri Nets and Process Algebra.* PhD thesis, Eindhoven University of Technology, Eindhoven, 1998.

[20] R. Bastide. Approaches in Unifying Petri Nets and the Object Oriented Approach. In *1st Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, 1995.

[21] R. Bastide and P. Palanque. Cooperative Objects : A Concurrent Petri Net Based Object-Oriented Language. In *IEEE System Man and Cybernetics 1993*, pages 286–291, Le Touquet, France, October 1993. Elseiver Science Publisher.

[22] E. Battiston, A. Chizzoni, and F. DeCindio. Inheritance and Concurrency in CLOWN. In *1st Workshop on Object-Oriented Programming Models Concurrency*, Turin, Italy, 1995.

[23] B. Baumgarten. *Petri-Netze: Grundlagen und Anwendungen.* Mannheim: BI-Wissenschaftsverlag, 1990.

[24] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60:109–137, 1984.

[25] L. Bernardinello and F. De Cindio. A Survey of Basic Net Models and Modular Net Classes. *Lecture Notes in Computer Science*, 609:304–351, 1992.

[26] G. Berthelot. Checking Properties of Nets using Transformations. In G. Rozenberg, editor, *Advances in Petri Nets 1985*, volume 222 of *Lecture Notes in Computer Science*, pages 19–40. Springer-Verlag, 1986.

[27] G. Berthelot. Transformations and Decompositions of Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Lecture Notes in Computer Science: Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 254, pages 359–376. Springer-Verlag, 1987.

[28] O. Biberstein. *CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems*. PhD thesis, University of Geneva, July 1997.

[29] O. Biberstein, D. Buchs, and N. Guelfi. Modeling of cooperative editors using COOPN/2. In G. Agha and F. de Cindio, editors, *Workshop on Object-Oriented Programming and Models of Concurrency'96*, Osaka, Japan, 1996.

[30] J. Billington. Many-sorted high level nets. In *3rd Workshop on Petri Nets and Performance Models, 11-13 December 1989*, pages 166–179, Washington, DC, USA, 1989. IEEE CS Press.

[31] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[32] G. Booch. *Object-Oriented Analysis and Design*. Benjamin Cummings, 1994 (2nd ed).

[33] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[34] J.P. Bowen and V. Stavridou. Safety-Critical Systems, Formal Methods and Standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[35] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On Behavioural Subtyping in LOTOS. In *Second IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 21–36. Chapman and Hall, 1997.

[36] W. Brauer, R. Gold, and W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. In *Advances in Petri Nets 1990*, pages 1–46, Berlin - Heidelberg - New York, November 1991. Springer.

[37] R. Brown. Frontline System Documentation, internal documentation, Hydro-Electric Corporation of Tasmania. 1997.

[38] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[39] J. Büchi. Weak second order logic and finite automata. *Z. Math. Logik, Grundlag. Math.*, 5:66–62, 1960.

[40] C. Capellmann, H. Dibold, and U. Herzog. Using High-Level Petri Nets in the Field of Intelligent Networks. In J. Billington, M. Diaz, and G. Rozenberg, editors, *Application of Petri nets to Communication Networks : Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science*, pages 37–68. Springer-Verlag, 1999.

[41] R. Cardell-Oliver. HTTDs and HOL. In C. Lewerentz and T. Lindner, editors, *Formal Development of Reactive Systems, Case Study Production Cell*, Lecture Notes in Computer Science, pages 261–276. Springer-Verlag, 1995.

[42] L. Cardelli and J.C. Mitchell. Operations on Records. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *The Conference on Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 75–81. Springer-Verlag, September 1989.

[43] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[44] D. Carrington, D.J. Duke, R. Duke, P. King, G.A. Rose, and G. Smith. Object-Z: An Object-Oriented Extension to Z. In S. Vuong, editor, *Formal Description Techniques, II (FORTE'89)*, pages 281–296. Elsevier Science Publishers (North-Holland), 1990.

[45] S. Caselli, G. Conte, and P. Marenzoni. Parallel State Space Exploration for GSPN Models. In G. De Michelis and M. Diaz, editors, *16th International Conference on Application and Theory of Petri Nets (ICATPN'95), Turin, June 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 181–200. Springer-Verlag, 1995.

[46] M. Ceska and V. Janousek. Object Orientation in Petri Nets. In *22nd Conference of the ASU*, pages 69–80, Clermont-Ferrand, France, 1996.

[47] S.C. Cheung and J. Kramer. Enhancing Compositional Reachability Analysis with Context Constraints. In *ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 115–125, December 1993.

[48] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed Coloured Nets and their Symbolic Reachability Graph. In *11th International Conference on Applications and Theory of Petri Nets (ICATPN'90), Paris, France, June 1990*, Lecture Notes in Computer Science, pages 387–411. Springer-Verlag, 1990.

[49] S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs. In M. Ajmone Marsan, editor, *14th International Conference on Application and Theory of Petri Nets 1993 (ICATPN'93), Chicago, Illinois, USA*, volume 691 of *Lecture Notes in Computer Science*, pages 186–205. Springer-Verlag, 1993.

[50] S. Christensen and L. Petrucci. Modular State Space Analysis of Coloured Petri Nets. In G. De Michelis and M. Diaz, editors, *16th International Conference on Application and Theory of Petri Nets (ICATPN'95), Turin, June 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, 1995.

[51] S. Christensen and L. Petrucci. Modular Analysis of Petri Nets. *The Computer Journal*, 43(3):224–242, 2000.

[52] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[53] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[54] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. In *The Fifth Workshop on Computer-Aided Verification*, pages 450–462, 1993.

[55] E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[56] P. Coad and E. Yourdon. *Object-Oriented Analysis, Second Edition.* Yourdon Press/Prentice Hall, 1991.

[57] W.R. Cook, W.L. Hill, and P.S. Canning. Inheritance Is Not Subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM SIGACT and SIGPLAN, ACM Press, 1990.

[58] Robert Corbett, Richard Stallman, and Wilfred Hansen. *Bison Manual: Using the YACC-compatible Parser Generator, for Version 1.29.* Free Software Foundation, Boston, MA, USA, November 1999.

[59] University of Aarhus CPN group, Computer Science Department. Petri Nets Tools Database.
⟨http://www.daimi.aau.dk/petrinet/tools/quick.html⟩.

[60] E. Cusack. Refinement, Conformance, and Inheritance. *Formal Aspects of Computing*, 3:129–141, January 1991.

[61] O. Dahl and K. Nygaard. SIMULA, an ALGOL-based Simulation Language. *Communications of the ACM*, 9(9):671–678, September 1966.

[62] C. Dalton. Cellular Encoding Extended and Applied to Real World Prediction Problems, internal report, Department of Computing, University of Tasmania. 2000.

[63] J. Desel. On Abstractions of Nets. In *Advances in Petri Nets 1991*, volume 524 of *Lecture Notes in Computer Science*, pages 78–92. 1991.

[64] J. Desel and A. Merceron. Vicinity respecting net morphisms. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 165–185. Springer-Verlag, November 1991.

[65] J. Desel and A. Merceron. Vicinity Respecting Homomorphisms for Abstracting System Requirements. Technical Report 337, Universität Karlsruhe, 1996.

[66] J. Desel and W. Reisig. Place/Transition Petri Nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Advances in Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 122–173. 1998.

[67] K. K. Dhara and G. T. Leavens. Forcing Behavioral Subtyping Through Specification Inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR95-20c.

[68] E.W. Dijkstra. Notes on Structured Programming. In *Structured Programming.* Academic Press, 1969.

[69] F. Emerson and A. Sistla. Symmetry and Model Checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, August 1996.

[70] J. Esparza. Decidability and Complexity of Petri Net Problems — An Introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets II: Advances in Petri Nets*, volume 1492 of *Lecture Notes in Computer Science*, pages 374–428. 1998.

[71] R. Esser. An Object Oriented Petri Net Language for Embedded System Design. In D. Budgen, G. Hoffnagle, and J. Trienekens, editors, *8th International Workshop on Software Technology and Engineering Practice*, pages 216–223. IEEE Computer Society Press, 1997.

[72] R. Fehling. Concept of Hierarchical Petri Nets with Building Blocks. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–168. Springer-Verlag, 1993.

[73] D.J. Floreani. *The Interconnection of Tactical Packet Radio Networks and BISDN.* PhD thesis, School of Physics and Electronic Systems Engineering, University of South Australia, 1996.

[74] D.J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. In J. Billington and W. Reisig, editors, *17th International Conference on Application and Theory of Petri Nets (ICATPN'97), Osaka, Japan, June, 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 1996.

[75] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.

[76] W. Garbe. The Petri Net Tools Survey. ⟨http://home.arcor-online.de/wolf.garbe/petrisoft.html⟩.

[77] H. J. Genrich, K. Lautenbach, and P. S. Thiagarajan. Elements of general net theory. In W. Brauer, editor, *Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg*, pages 21–163, Berlin, FRG, 1979. Springer. Appeared as Lecture Notes in Computer Science 84.

[78] H.J. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Bad Honnef, September 1986*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[79] P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1996.

[80] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[81] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[82] S. Gordon and J. Billington. Analysing a Missile Simulator using Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):144–159, December 1998.

[83] S. Gordon and J. Billington. Applying Coloured Petri nets and Design/CPN to an Air-to-Air Missile Simulator. In K. Jensen, editor, *Proceedings of Workshop on Practical use of Coloured Petri Nets and Design/CPN*, pages 1–14. Department of Computer Science, Aarhus University, June 1998.

[84] I. Graham, B. Hendersen-Sellers, and H. Younessi. *The OPEN Process Specification.* Addison-Wesley, 1997.

[85] C.A. Gurr. Supporting Formal Reasoning for Safety-Critical Systems. *High Integrity Systems*, 1(4):386–396, 1996.

[86] S. Haddad. A Reduction Theory for Coloured Nets. In *Advances in Petri Nets 1989*, pages 209–235. Springer-Verlag, June 1989.

[87] J.A. Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.

[88] D. Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987.

[89] P. Harmon, D.A. Taylor, and W. Morrissey. *Objects in Action : Commercial Applications of Object-Oriented Technologies.* Monographs in Theoretical Computer Science. Addison-Wesley, Reading, MA, 1993.

[90] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[91] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International Series in Computer Science, 1985.

[92] C.M. Holloway. Why Engineers Should Consider Formal Methods. In *16th AIAA/IEEE Digital Avionics Systems Conference*, pages 1.3–16 – 1.3–22, Irvine, California, 1997.

[93] G.J. Holzmann. Automated Protocol Validation in *Argos*: Assertion Proving and Scatter Searching. *IEEE Transactions on Software Engineering*, 13(6):683–696, June 1987.

[94] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[95] P. Huber, A.M. Jensen, L.O. Jepsen, and K. Jensen. Reachability Trees for High-Level Petri Nets. *Theoretical Computer Science*, 45(3):261–292, 1986.

[96] N. Husberg. Maria - a Modular Reachability Analyzer. Theoretical Computer Science Laboratory, Helsinki University of Technology, Research plan 4.0, 1998.

[97] Hans Hüttel. *Decidability, Behavioural Equivalences and Infinite Transition Graphs.* Ph.D. thesis, Computer Science Dept., University of Edinburgh, December 1991.

[98] ISO. *ISO 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* Geneva, Switzerland, 15 February 1987.

[99] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Publishing Company, March 1994.

[100] I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[101] J.W. Janneck and M. Naedele. Modeling a Die Bonder with Petri Nets — A Case Study. *IEEE Transactions on Semiconductor Manufacturing*, 11(3):404–409, 1998.

[102] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[103] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1995.

[104] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use: Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[105] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN^{TM}: A Reference Manual.* MetaSoftware Corporation, 1992.

[106] J. Jørgensen. Analysing Coloured Petri Nets by the Occurrence Graph Method. Technical Report PB-517, Department of Computer Science, University of Aarhus, February 1997.

[107] P. Kemper. Reachability Analysis Based on Structured Representations. In J. Billington and W. Reisig, editors, *17th International Conference on Application and Theory of Petri Nets (ICATPN'97), Osaka, Japan, June, 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 269–288. Springer-Verlag, 1996.

[108] E. Kindler and W. Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, (51):16–31, December 1996.

[109] E. Kindler and H. Völzer. Flexibility in Algebraic Nets. volume 1420 of *Lecture Notes in Computer Science*, pages 345–364. Springer-Verlag, June 1998.

[110] L.M. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In J. Desel and M. Silva, editors, *19th International Conference on Application and Theory of Petri Nets (ICATPN'98), Lisbon, Portugal, June 1998*, volume 1420 of *Lecture Notes in Computer Science*, pages 104–123. Springer-Verlag, June 1998.

[111] B.R. Ladeau and C.W. Freeman. Using Formal Specification for Product Development. *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, 42(5):46–50, December 1991.

[112] C. A. Lakos. Pragmatic Inheritance Issues for Object Petri Nets. In *TOOLS Pacific, Melbourne, Australia 1995*, pages 309–321. Prentice-Hall, 1995.

[113] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In G. De Michelis and M. Diaz, editors, *16th International Conference on Application and Theory of Petri Nets (ICATPN'95), Turin, June 1995*, volume 935 of *Lecture Notes in Computer Science*, pages 278–297, 1995.

[114] C.A. Lakos. A Cooperative Editor for Hierarchical Diagrams: An Object Petri Net Model. In *2nd Workshop on Object-Oriented Programming and Models of Concurrency*, Osaka, Japan, 1996.

[115] C.A. Lakos. The LOOPN++ User Manual. Technical Report TR96-1, Computer Science Department, University of Tasmania, 1996.

[116] C.A. Lakos. Composing Abstractions of Coloured Petri Nets. In *21st International Conference on Application and Theory of Petri Nets (ICATPN'2000) Aarhus, Denmark, June 25-29, 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 323–342. Springer-Verlag, June 2000.

[117] C.A. Lakos. Re: Proof of Bisimilarity, Wed 6 December 2000. Personal correspondence (email).

[118] C.A. Lakos. Meeting, March 2001. Personal correspondence.

[119] C.A. Lakos. Re: A Quick Question, Thu 25 January 2001. Personal correspondence (email).

[120] C.A. Lakos. Re: Further notes, May 2001. Personal correspondence (email).

[121] C.A. Lakos. Re: Minimally complete optimisation (ctd), Mar 2001. Personal correspondence (email).

[122] C.A. Lakos and J. Lamp. The Incremental Modelling of the Z39.50 Protocol with Object Petri Nets. In J. Billington, M. Diaz, and G. Rozenberg, editors, *Application of Petri nets to Communication Networks : Advances in Petri Nets*, volume 1605 of *Lecture Notes in Computer Science*, pages 37–68. Springer-Verlag, 1999.

[123] C.A. Lakos, J. Lamp, C. Keen, and B. Marriott. Modelling Network Protocols with Object Petri Nets. In *Workshop on Petri Nets Applied to Protocols*, pages 31–42, 1995.

[124] C.A. Lakos and G.A. Lewis. A Practical Approach to Incremental Specification. In *IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems*. Kluwer, 2000.

[125] C.A. Lakos and G.A. Lewis. Behaviour Inheritance for Object Lifecycles. In *33rd International Conference on Technology of Object-Oriented Languages and Systems - Europe TOOLS33 2000, Mont Saint-Michel, France*, pages 262–276. IEEE Computer Society Press, June 2000.

[126] W. LaLonde and J. Pugh. Subclassing $\neq$ Subtyping $\neq$ Is-a. *Journal of Object-Oriented Programming*, pages 57–60, January 1991.

[127] J.W. Lamp. Encoding the ANSI Z39.50 Search and Retrieval Protocol using LOOPN. Honours thesis, Computer Science Department, University of Tasmania, 1994.

[128] G.T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, MIT, December 1988. Published as MIT/LCS/TR-439 in February 1989.

[129] G.A. Lewis and C.A. Lakos. Incremental State Space Analysis of Coloured Petri Nets. In *22nd International Conference on Application and Theory of Petri Nets (ICATPN'2001) Newcastle upon Tyne, United Kingdom, June 25-29, 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 263–282, June 2001.

[130] M. Lindqvist. Parameterized Reachability Trees for Predicate/Transition Nets. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 301–324. Springer-Verlag, 1993.

[131] B. Liskov. Keynote Address — Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1987.

[132] B. Liskov and J.M. Wing. A New Definition of the Subtype Relation. In O.M. Nierstrasz, editor, *7th European Conference on Object-Oriented Programming (ECOOP'93), Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, NY, July 1993.

[133] B.H. Liskov and J.M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[134] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996.

[135] D. Lütkehaus and A. Zeller. *DDD — Data Display Debugger*. Free Software Foundation, Boston, MA, USA, 1999.

[136] M. Mäkelä. *Maria — Modular Reachability Analyzer for Many-Sorted Petri Nets*. Helsinki University of Technology, Espoo, Finland, 1999. version 0.1.

[137] M. Mäkelä. A Reachability Analyser for Algebraic System Nets. Licentiate thesis, Helsinki University of Technology, Theoretical Computer Science Laboratory, Espoo, Finland, March 2000.

[138] J. Martin and J. Odell. *Object-Oriented Analysis & Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[139] B. Meyer. Lessons From the Design of the Eiffel Libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[140] B Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, second edition, 1997.

[141] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[142] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[143] A.B. Mnaouer, T. Sekiguchi, Y. Fujii, and T. Ito. Coloured Petri Nets Based Modelling and Simulation of the Static and Dynamic Allocation Policies of the Asynchronous Bandwidth in the Fieldbus Protocol. In *Application of Petri nets to Communication Networks : Advances in Petri Nets*, volume 1605, pages 93–130. 1999.

[144] L.E. Moser and P.M. Melliar-Smith. Formal Verification of Safety-Critical Systems. *Software Practice and Experience*, 20(8):799–821, August 1990.

[145] T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[146] R.M. Needham and M. Schroeder. Using Encryption for Authentication in Large Computer Networks. *Communications of the ACM*, 12(21):993–999, 1978.

[147] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96(1):3–33, April 1992.

[148] O. Nierstrasz. Regular Types for Active Objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993. Proceedings of the 8th. annual conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93, Washington DC, 1993.

[149] E. Nuutila and E. Soisalon-Soininen. On Finding the Strongly Connected Components in a Directed Graph. *Information Processing Letters*, 49(1):9–14, January 1994.

[150] Object Management Group. CORBA 2.0/IIOP specification. Technical Report PTC/96-03-04, Framingham Corporate Center, Framingham (MA), USA, 1996.

[151] J. Padberg, M. Gajewsky, and C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 221–238, 1998.

[152] Parasoft. *Insure++ — A Tool to Support Total Quality Software*, 1996. ⟨http://www.parasoft.com⟩.

[153] E. Pastor, O. Roig, J. Cortadella, and R.M. Badia. Petri Net Analysis Using Boolean Manipulation. In R. Valette, editor, *15th International Conference on Application and Theory of Petri Nets (ICATPN'94),Zaragoza, Spain, June 20-24, 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 416–435. Springer-Verlag, 1994.

[154] V. Paxson, V. Jacobson, J. Poskanzer, and K. Gong. *Flex: The Lexical Scanner Generator, for Version 2.5*. Free Software Foundation, Boston, MA, USA, April 1995.

[155] D. Peled. Combining Partial Order Reductions with On-the-fly Model-checking. In D. L. Dill, editor, *1994 Workshop on Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994.

[156] B. Perens. *Electric Fence Malloc Debugger*. Pixar Animation Studios, 1993. Manual page for Version 2.0.5.

[157] R.V. Peri. *Specification and Verification of Security Policies*. PhD thesis, University of Virginia, January 1996. ⟨ftp://ftp.cs.virginia.edu/pub/dissertations/9603.ps.Z⟩.

[158] C. Petri. *Kommunikation mit Automaten.* Dissertation, Technische Universität Darmstadt, Fachbereich Mathematik, Physik, Darmstadt, Germany, 1962.

[159] C.A. Petri. Introduction to General Net Theory. In W. Brauer, editor, *Net Theory and Applications, Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg, 1979*, volume 84 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 1980.

[160] L. Pomello, G. Rozenberg, and C. Simone. A Survey of Equivalence Notions for Net Based Systems. In G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 410–472. Springer-Verlag, 1992.

[161] A. Ralston and E.D. Reilly, editors. *Encyclopedia of Computer Science.* International Thomson Computer Press, 3rd edition, 1995.

[162] W. Reisig. *Petri Nets (An Introduction).* Number 4 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[163] W. Reisig. Petri Nets in Software Engineering. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 63–96. Springer-Verlag, 1987.

[164] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991.

[165] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science.* 1998.

[166] S. Roch and P.H. Starke. INA Integrated Net Analyzer Version 2.2 Manual. electronic, 2000.
⟨http://www.informatik.hu-berlin.de/lehrstuehle/automaten/ina/#manual⟩.

[167] P. Rook, editor. *Software Reliability Handbook.* Elsevier Applied Science, 1990.

[168] G. Rozenberg and J. Engelfriet. Elementary Net Systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 12–121. Springer-Verlag, 1998.

[169] S. Rudkin. Inheritance in LOTOS. In K. Parker and G. Rose, editors, *Formal Description Techniques, IV, Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '91, Sydney, Australia, 19-22 November 1991*, pages 409–424, 1991.

[170] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design.* Prentice-Hall Inc., 1991.

[171] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1999.

[172] J. Rushby. Formal Specification and Verification of a Fault–Masking and Transient–Recovery Model for Digital Flight–Control Systems. In J. Vyptopil, editor, *Proceedings of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 237–258. Springer-Verlag, January 1991.

[173] M. Sakkinen. Disciplined Inheritance. In S. Cook, editor, *3rd European Conference on Object-Oriented Programming (ECOOP'89)*, British Computer Society Workshop Series, pages 39–56. Cambridge University Press, 1989.

[174] S.M. Shatz, S. Tu, T. Murata, and S. Duri. Application of Petri Net Reduction for Ada Tasking Deadlock Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307–1322, December 1996.

[175] M. Shaw. Patterns for Software Architectures. In J. Coplien and D. Schmidt, editors, *Pattern Languages of Program Design*, chapter 24, pages 453–462. Addison-Wesley, 1995.

[176] S. Shlaer and S.J. Mellor. *Object Oriented Life Cycles: Modeling the World in States*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1992.

[177] C. Sibertin-Blanc. Cooperative Nets. In R. Valette, editor, *15th International Conference on Application and Theory of Petri Nets (ICATPN'94), Zaragoza, Spain, June 20-24, 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490. Springer-Verlag, 1994.

[178] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. *ACM SIGPLAN Notices*, 21(11):38–45, November 1986. OOPSLA '86 Conference Proceedings, N. Meyrowitz (editor), September 1986, Portland, Oregon.

[179] S. So-Ming and M. Ito. GrafTab - An Innovative Requirements Specification Method (A Requirements Specification of a Steam Boiler Controller). Technical report, 1995.
⟨http://www.informatik.uni-kiel.de/procos/dag9523/steam-boiler-solutions.html⟩.

[180] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Boston, MA, USA, 1999. Software Version 2.95.

[181] P.H. Starke. Reachability Analysis of Petri Nets Using Symmetries. *Systems Analysis Modeling Simulation*, 8(4):293–303, 1991.

[182] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, 2000. First print of an unpublished manuscript written 1967.

[183] A. Taivalsaari. On the Notion of Inheritance. *Computing Surveys*, 28(3):438–479, September 1996.

[184] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[185] P.S. Thiagarajan. Elementary Net Systems. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part I, Bad Honnef, September 1986*, volume 254 of *Lecture Notes in Computer Science*, pages 26–59. Springer-Verlag, 1987.

[186] M. Tiusanen. Symbolic, Symmetry, and Stubborn Set Searches. In R. Valette, editor, *15th International Conference on Application and Theory of Petri Nets (ICATPN'94),Zaragoza, Spain, June 20-24, 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 511–530, 1994.

[187] L. Torvalds. Linux: a portable operating system. Master's thesis C-1997-12, University of Helsinki, Department of Computer Science, Helsinki, Finland, 1997.

[188] M. Trompedeller. A Petri Net Classification and Related Tools. ⟨http://twilight.dsi.unimi.it/Users/Tesi/trompede/petri/⟩.

[189] W.T. Tuttle. *Graph Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2001.

[190] A. Valmari. Stubborn Sets of Coloured Petri Nets. In *12th International Conference on Application and Theory of Petri Nets (ICATPN'91), June 1991, Gjern, Denmark*, Lecture Notes in Computer Science, pages 102–121. Springer-Verlag, 1991.

[191] A. Valmari. Compositional State Space Generation. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 427–457. Springer-Verlag, 1993.

[192] A. Valmari. Compositional Analysis with Place-Bordered Subnets. In R. Valette, editor, *15th International Conference on Application and Theory of Petri Nets (ICATPN'94),Zaragoza, Spain, June 20-24, 1994*, volume 815 of *Lecture Notes in Computer Science*, pages 531–547, 1994.

[193] A. Valmari. Compositionality in State Space Verification Methods. In J. Billington and W. Reisig, editors, *17th International Conference on Application and Theory of Petri Nets (ICATPN'97), Osaka, Japan, June, 1996*, volume 1091 of *Lecture Notes in Computer Science*, pages 29–56, 1996.

[194] A. Valmari. The State Explosion Problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[195] R.J. van Glabbeek. The Linear Time-Branching Time Spectrum (Extended Abstract). In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, The Netherlands, August 1990. Springer-Verlag.

[196] R.J. van Glabbeek. The Linear Time–Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves (Extended Abstract). In E. Best, editor, *CONCUR'93: 4th International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, August 1993. Springer-Verlag.

[197] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995.

[198] G. Watson. *Debug Malloc Library Version 4.7.1*, 2000. ⟨http://dmalloc.com⟩.

[199] P. Wegner. Dimensions of Object-Based Language Design. *ACM SIGPLAN Notices*, 22(12):168–182, December 1987.

[200] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[201] P. Wegner and S.B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *2nd European Conference on Object-Oriented Programming 1988 (ECOOP'88), Oslo, Norway*, volume 322 of *Lecture Notes in Computer Science*, pages 55–77, Oslo, Norway, 1988. Springer-Verlag.

[202] H. Wehrheim. Behavioural Subtyping and Property Preservation. In *IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 213–232. Kluwer, 2000.

[203] G. Winskel. Petri Nets, Algebras, Morphisms, and Compositionality. *Information and Computation*, 72(3):197–238, March 1987.

[204] P. Wolper and P. Godefroid. Partial-Order Methods for Temporal Verification. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, Germany, August 1993. Springer-Verlag.

[205] W. Yeh. *Controlling State Explosion in Reachability Analysis*. PhD thesis, Department of Computer Sciences, Purdue University, West Lafayette, 1993.

[206] W.J. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 49–59. ACM Press, New York, USA, 1991.